Sendmail X:
Requirements,
Architecture,
Functional Specification,
Implementation,
and Performance

Claus Assmann

May 4, 2005

Contents

1 Sendmail X: Requirements **19**

1.1 Requirements . . . 19

1.1.1 Basic Requirements . . . 19

1.1.2 Other Requirements . . . 20

1.1.3 Explanation of Buzzwords . . . 20

1.1.3.1 Robustness . . . 21

1.1.3.2 Flexibility . . . 21

1.1.3.3 Scalability . . . 21

1.1.3.4 Extendability . . . 21

1.1.3.5 Maintainability . . . 21

1.1.3.6 Portability . . . 22

1.2 Restrictions . . . 22

1.3 Configuration . . . 22

1.3.1 Configuration File . . . 22

1.3.2 Configuration Options . . . 23

1.4 Documentation . . . 23

1.4.1 Documentation Format . . . 24

2 Sendmail X: Architecture **25**

2.1 General Architecture of sendmail X . . . 25

2.1.1 Overview . . . 25

2.1.2 Remarks about Performance . . . 26

2.1.3 Remarks about Configuration . . . 27

2.1.3.1 Simplicity and Flexibility . . . 27

2.1.3.2 Option Grouping . . . 27

2.1.3.3 Accessing and Changing the Configuration . . . 28
2.1.4 Security Considerations . . . 28
2.1.5 Control and Data Flow . . . 28
2.2 Configuration . . . 29
2.2.1 General . . . 29
2.2.2 Syntax . . . 30
2.2.2.1 Comments . . . 31
2.2.2.2 Internationalization . . . 32
2.2.2.3 Strings and other Types . . . 32
2.2.2.4 Inclusion . . . 32
2.2.3 Alternative Configuration Formats . . . 32
2.2.4 Configurability . . . 33
2.2.4.1 Consistency . . . 36
2.2.5 Configuration File Structure . . . 36
2.2.6 Option Names . . . 38
2.2.7 Dynamic Configuration . . . 38
2.2.7.1 Configuration: Conditionals . . . 39
2.2.7.2 Configuration: Reference to Maps . . . 39
2.2.8 Default Configuration . . . 40
2.2.9 Displaying Current Configuration . . . 40
2.2.10 Dynamically Changing Configuration . . . 40
2.2.11 Configuration Example for PreAlpha Version . . . 41
2.2.11.1 Anti-Spam Configuration for PreAlpha Version . . . 43
2.2.12 Security Considerations . . . 43
2.3 Supervisor . . . 44
2.3.1 Security Considerations . . . 44
2.4 Queue Manager . . . 44
2.4.1 Queues . . . 45
2.4.1.1 Current Selection of Queues and their Names . . . 47
2.4.2 Queue Formats . . . 47
2.4.2.1 Reference Counting . . . 47
2.4.3 Data Flow: Moving Envelopes between Queues . . . 48
2.4.3.1 Terminology . . . 48

2.4.3.2 Data Flow: Queue oriented View 49
2.4.3.3 Detailled Data Flow: Transaction/Recipient oriented View 50
2.4.3.4 Updating Data in Queues after a Delivery Attempt 51
2.4.3.5 Reading Entries from Deferred Queue 52
2.4.3.6 Cut-Through Delivery 53
2.4.4 Scheduling 54
2.4.4.1 Two Level Scheduling 56
2.4.4.2 First Level Scheduling (Global) 58
2.4.4.3 Second Level Scheduling (Micro) 58
2.4.4.4 Minimizing Number of Transactions 59
2.4.4.5 Cleanup of Active Queue 59
2.4.4.6 Outgoing Connection Cache 59
2.4.4.7 Multiple Destinations 59
2.4.4.8 Data Structures to support Scheduling 60
2.4.5 Triggering Deliveries 61
2.4.6 DSNs 61
2.4.6.1 DSN Recipient Types 62
2.4.6.2 DSN: Return Body or Headers 63
2.4.6.3 DSN: Delayed 63
2.4.7 Load Control 63
2.4.8 Misc 65
2.4.9 Security Considerations 65
2.5 SMTP Server Daemon 66
2.5.1 Internet Server Application 66
2.5.2 SMTP Server Design Alternatives 67
2.5.3 Control Flow 69
2.5.4 Anti-Spam Checks 70
2.5.5 Valid User Checks 71
2.5.6 Address Rewriting 71
2.5.6.1 Envelope Address Rewriting 71
2.5.6.2 Header Address Rewriting 71
2.5.7 Security Considerations 71
2.6 Address Resolver 72

2.6.1 Address Resolver Operation . . . 72
2.6.2 Generic Comment about Map Lookups . . . 72
2.6.3 Mail Routing . . . 73
2.6.4 Address Rewriting . . . 73
2.6.5 Proposal for Routing and Rewriting . . . 73
2.6.6 Valid Recipients . . . 74
2.6.6.1 Valid Local Recipients . . . 74
2.6.6.2 Valid Remote Recipients . . . 74
2.6.7 Aliases . . . 75
2.6.7.1 Forward . . . 76
2.6.7.2 Other Approaches to Aliasing? . . . 76
2.6.8 Expansion to Multiple Addresses . . . 77
2.6.9 Virtual Hosting . . . 77
2.6.10 Fallback . . . 77
2.6.11 Security Considerations . . . 77
2.7 Initial Mail Submission . . . 77
2.7.1 Initial Mail Submission Alternatives . . . 78
2.7.1.1 How to use queue directory? . . . 78
2.7.1.2 Misc . . . 79
2.7.2 Security Considerations . . . 79
2.8 Mail Delivery Agents . . . 79
2.8.1 Delivery Agent Modules . . . 79
2.8.2 Specifying Delivery Agents . . . 80
2.8.3 Local Delivery Agent . . . 80
2.8.4 Security Considerations . . . 81
2.9 SMTP Client . . . 81
2.9.1 Control Flow . . . 82
2.9.2 Security Considerations . . . 82
2.10 Milter . . . 83
2.10.1 Possible Enhancements . . . 83
2.10.2 More Ideas for Enhancements . . . 83
2.10.3 Security Considerations . . . 83
2.11 Miscellaneous Programs . . . 84

2.11.1 Access to the Queue Manager . . . 84
2.11.1.1 Show Mail Queue . . . 84
2.11.1.2 Force Queue Run . . . 84
2.11.1.3 Mailstats . . . 84
2.11.1.4 Performance Statistics . . . 84
2.11.2 Security Considerations . . . 84
2.12 Maps . . . 84
2.13 Modules . . . 86
2.13.1 Security Considerations . . . 86
2.14 Security Hints . . . 86
2.14.1 Privileged Access . . . 87
2.14.1.1 Access to Files . . . 88
2.14.1.2 Sockets . . . 88
2.14.1.3 Running as a different user . . . 88
2.15 Misc . . . 89
2.15.1 Misc Misc . . . 89
2.15.2 Configuration . . . 89
2.15.3 Configuration Changes . . . 89
2.15.4 Performance Measurements: Profiling . . . 90
2.15.5 Logging . . . 90
2.15.5.1 Logging Granularity . . . 90
2.15.5.2 Parsing Logfiles . . . 91
2.15.5.3 Extensible Logging . . . 92
2.15.6 Debugging . . . 92
2.15.7 Robust Programming . . . 92
2.16 Schedule . . . 92
2.17 Glossary . . . 93

3 Sendmail X: Functional Specification 95
3.1 Functional Specification of sendmail X . . . 95
3.1.1 Asynchronous APIs . . . 95
3.1.1.1 Generic Description of Asynchronous Operation . . . 96
3.1.1.2 Some Remarks about Identifiers . . . 96
3.2 Configuration . . . 97

3.2.1 Configuration Structure . . . 97
3.2.2 Naming Conventions . . . 97
3.2.3 Specification of Classes (Lists, Sets) . . . 98
3.3 Supervisor . . . 98
3.3.1 External Interfaces . . . 98
3.3.2 Operation . . . 99
3.3.3 Shutdown . . . 99
3.3.4 Configuration . . . 100
3.3.5 Internal Interfaces . . . 102
3.4 Queue Manager . . . 102
3.4.1 External Interfaces . . . 103
3.4.2 Shutdown . . . 103
3.4.3 Internal Interfaces . . . 103
3.4.4 Indices for Accessing the EDBs . . . 103
3.4.5 Interface to SMTP DAs . . . 104
3.4.5.1 QMGR - Delivery Agents API . . . 104
3.4.6 Transferring Data between EDBs . . . 106
3.4.6.1 Detailled Data Flow for Cut-Through Delivery . . . 107
3.4.6.2 Reading Entries from Deferred Queue . . . 107
3.4.7 Reconstructing Data from IBDB . . . 108
3.4.7.1 Cleaning up IBDB . . . 109
3.4.8 Deferred Envelope Database: Recipient Addresses . . . 109
3.4.9 Scheduler Algorithms . . . 110
3.4.9.1 Slow Start and Connection Limit . . . 110
3.4.10 Data Structures . . . 111
3.4.10.1 Connection Cache Access . . . 111
3.4.10.2 Connection Reuse Problem . . . 111
3.4.10.3 Data for DSN . . . 112
3.4.10.4 Incoming Queue (INCEDB) . . . 112
3.4.10.5 Active Queue (ACTEDB, AQ) . . . 114
3.4.10.6 Deferred Queue (DEFEDB) . . . 117
3.4.10.7 Deferred Queue Cache (EDBC) . . . 119
3.4.10.8 Connection Cache (incoming) . . . 119

3.4.10.9 Connection Status (incoming) . . . 119
3.4.10.10 Connection Cache (outgoing) Connections . . . 119
3.4.10.11 Connection Status (outgoing) . . . 121
3.4.10.12 DA Status Cache . . . 121
3.4.10.13 Load Control Data Structures . . . 122
3.4.11 Database and Cache APIs . . . 123
3.4.12 QMGR - SMTPS API . . . 123
3.4.13 QMGR - First Level Scheduler API . . . 124
3.4.14 QMGR - Delivery Scheduler API . . . 124
3.4.15 Interface to AR . . . 125
3.4.15.1 Aliases . . . 125
3.4.16 Updating Data in Queues after a Delivery Attempt . . . 126
3.4.16.1 Preserving Order of Updates . . . 127
3.4.17 Load Control Functionality . . . 128
3.4.17.1 Load Control Functionality: Throttling . . . 129
3.4.17.2 Load Control Functionality: Unthrottling . . . 130
3.4.17.3 Handling out of Memory . . . 130
3.4.18 Manual Interaction with QMGR . . . 130
3.4.18.1 Getting Status Informations . . . 131
3.4.18.2 Triggering Actions . . . 131
3.5 SMTP Server Daemon . . . 131
3.5.1 External Interfaces . . . 131
3.5.2 Control Flow . . . 131
3.5.2.1 Startup . . . 132
3.5.2.2 States . . . 132
3.5.2.3 Data Structures . . . 133
3.5.2.4 Detailled Control Flow . . . 134
3.5.2.5 Pipelining . . . 136
3.5.2.6 Anti-Spam Checks . . . 137
3.5.2.7 Anti-Spam Checks: Functionality . . . 142
3.5.2.8 Anti-Spam Checks: Functionality for first Release . . . 142
3.5.2.9 Anti-Spam Checks: API . . . 144
3.5.2.10 Valid Recipient Checks . . . 145

3.5.2.11 Valid Sender Checks 145

3.5.2.12 Map Lookups 146

3.5.3 Policy Milter Interface Behavior 146

3.5.3.1 Interaction between Anti-Spam Checks and Milter 147

3.5.4 More Generic Return and Reply Code Behavior 147

3.5.5 Internal Interfaces 148

3.5.5.1 Throttling 148

3.5.5.2 Removal of Entries from CDB 148

3.6 Address Resolver 149

3.6.1 External Interfaces 149

3.6.2 Valid Recipients 149

3.6.2.1 Valid Local Recipients 149

3.6.3 Internal Interfaces 150

3.6.3.1 Data Structures 150

3.6.3.2 Error Behavior 151

3.6.3.3 Caching of Map Lookups 151

3.6.3.4 Threading Model 151

3.7 Initial Mail Submission 152

3.7.1 External Interfaces 152

3.7.2 Internal Interfaces 152

3.8 Mail Delivery Agents 152

3.8.1 External Interfaces 152

3.8.2 Configuration 152

3.8.3 Configuring Multiple DAs 155

3.8.3.1 Selecting and Naming Delivery Agents 155

3.8.4 Internal Interfaces 157

3.8.4.1 Status Information from DA to QMGR 157

3.9 SMTP Client 158

3.9.1 External Interfaces 158

3.9.2 Internal Interfaces 158

3.9.2.1 Status Information from SMTP Client to QMGR 158

3.10 Milter 159

3.10.1 External Interfaces 159

3.10.2 Internal Interfaces . . . 159
3.11 Miscellaneous Programs . . . 159
3.11.1 Show Mail Queue . . . 159
3.11.2 Force Queue Run . . . 160
3.11.3 Mailstats . . . 160
3.12 Security Hints . . . 160
3.12.1 Owners and Permissions . . . 160
3.13 Databases and Caches for Envelopes, Contents, and Connections . . . 161
3.13.1 DBs with Multiple Access Keys . . . 161
3.13.2 DBs with Non-unique Access Keys . . . 161
3.13.3 Envelope Database Access Methods . . . 162
3.13.4 Incoming Envelope Database API . . . 162
3.13.4.1 Incoming Envelope Database API: RSC . . . 163
3.13.4.2 Incoming Envelope Database: Misc . . . 164
3.13.4.3 Incoming Envelope Database API: Disk Backup . . . 165
3.13.5 Active Envelope Database API . . . 166
3.13.6 Deferred Envelope Database API . . . 167
3.13.6.1 Deferred Envelope Database Implementation . . . 169
3.13.7 Content Database . . . 171
3.13.7.1 Content Database API . . . 172
3.13.7.2 CDB: Group Commits . . . 173
3.13.7.3 Logging as CDB Implementation . . . 173
3.13.7.4 CDB: Meta Data Operations . . . 174
3.13.7.5 CDB: File Reuse . . . 174
3.13.8 Connection Database (Cache) . . . 175
3.13.9 Available Database Implementation . . . 175
3.13.10 Restricted Size Caches . . . 175
3.13.10.1 Restricted Size Caches With Varying Size . . . 176
3.13.11 Atomic Updates . . . 176
3.13.11.1 Atomic Disk Writes . . . 177
3.13.12 Data Structures for Queues on Disk and for Communication . . . 177
3.13.12.1 Size Estimates . . . 178
3.13.13 Misc . . . 179

3.14 Maps . . . 179
3.14.1 Results of Map Lookups . . . 179
3.14.2 Lookup Functionality . . . 179
3.14.2.1 Placing the '@' sign . . . 180
3.14.3 Replacing Patterns in RHS . . . 181
3.15 Error Handling . . . 181
3.15.1 Returning Errors . . . 181
3.15.2 Error Classification . . . 181
3.15.3 Converting Error Codes into Textual Descriptions . . . 182
3.15.4 Misc . . . 182
3.16 Libraries . . . 182
3.16.1 Naming Conventions . . . 183
3.16.1.1 Naming Conventions for Types . . . 183
3.16.1.2 Naming Conventions for Functions . . . 183
3.16.1.3 Naming Conventions for Variables . . . 183
3.16.1.4 Naming Conventions for Structures . . . 184
3.16.1.5 Naming Conventions for Macros . . . 184
3.16.2 Include Files . . . 184
3.16.3 I/O . . . 185
3.16.3.1 Required I/O Functionality . . . 185
3.16.3.2 I/O Buffer . . . 185
3.16.4 Event Driven Loop . . . 186
3.16.5 Resource Pools . . . 187
3.16.6 Memory Handling . . . 187
3.16.7 String Abstraction . . . 187
3.16.7.1 Constant Strings . . . 188
3.16.8 Timed Events . . . 189
3.16.9 Shared Memory . . . 189
3.16.10 Address Parsing . . . 189
3.16.10.1 RFC 2821 parsing . . . 190
3.16.10.2 RFC 2822 parsing . . . 191
3.16.10.3 Token handling . . . 192
3.16.10.4 Address rewrite engine . . . 192

3.16.11 Internal Communication . . . 192

3.16.11.1 Data Structures for Internal Communication . . . 192

3.16.11.2 Marking End of Record . . . 195

3.16.11.3 A Note about Record Types . . . 195

3.16.12 Common API for Storage Libraries (Maps) . . . 196

3.16.12.1 Data Structure for Storage Libraries . . . 198

3.16.12.2 Abstraction Layer for Storage Libraries . . . 199

3.16.12.3 Dynamic (Re)Open of Maps . . . 204

3.16.12.4 Asynchronous Operation of Maps . . . 204

3.16.12.5 Multi-Threading . . . 204

3.16.12.6 Multiple Map Instances . . . 205

3.16.13 DNS . . . 205

3.16.14 Timeouts . . . 206

3.16.15 Misc . . . 207

3.16.16 Logging . . . 208

3.16.16.1 ISC Logging . . . 208

3.17 Modules . . . 209

3.18 Building sendmail X . . . 210

3.18.1 autotools . . . 210

3.19 Operating System Calls . . . 210

3.19.1 General Hints . . . 210

3.19.2 Operating System Specific Hints . . . 211

3.20 Worker Threads . . . 211

3.20.1 SMTP Server and Worker Threads . . . 212

3.20.2 Worker Thread Library . . . 212

3.20.3 Thread Libraries . . . 213

3.20.3.1 State Threads for Internet Applications . . . 213

3.20.4 Dealing with Blocking Sections for Worker Threads . . . 213

3.20.4.1 Worker Threads with Semaphores . . . 214

3.20.4.2 Worker Threads with Dynamic Limits . . . 214

3.20.4.3 Counting Active Threads . . . 214

3.20.4.4 Summary: Dealing with Blocking Sections for Worker Threads . . . 215

3.20.5 Event Thread Library . . . 215

3.20.5.1 Overview 215

4 Sendmail X: Implementation 219

4.1 Introduction to the Sendmail X Source Code 219

4.1.1 Identifiers 219

4.1.2 Asynchronous Functions 220

4.1.2.1 Two Different Problems 220

4.1.2.2 Two Approaches 220

4.1.2.3 Asynchronous Functions: Callback Synchronization 222

4.1.2.4 Asynchronous Functions: Result Queue 224

4.1.3 Transaction Based Processing 225

4.1.4 Secure Programming 226

4.2 Schedule 226

4.3 Libraries 227

4.3.1 Queues, Lists, etc 228

4.3.2 Hash Tables 228

4.3.3 Classes 228

4.3.4 Trees 228

4.3.5 RSC 228

4.3.5.1 Typed RSC 229

4.3.6 DBs with Multiple Access Keys 230

4.3.7 DBs with Non-unique Access Keys 230

4.3.8 Event Thread Library 230

4.3.9 RCB Communication 232

4.3.10 DNS 233

4.3.10.1 DNS Data Structures 233

4.3.10.2 DNS Functions 235

4.3.11 Logging 236

4.3.11.1 Logfile Rotation 236

4.4 SMTP Server Daemon 237

4.4.1 First Prototype 237

4.4.1.1 Misc 237

4.4.2 Communication between SMTPS and QMGR 237

4.4.2.1 Implementation of Communication between SMTPS and QMGR 238

4.4.2.2 SMTPS - QMGR Protocol . . . 238

4.4.2.3 SMTPS - SMAR Protocol . . . 239

4.5 Mail Delivery Agents . . . 240

4.5.1 Maintaining Delivery Classes and Agents . . . 240

4.6 SMTP Client . . . 241

4.6.1 First Prototype . . . 241

4.6.1.1 SMTPC - QMGR Protocol . . . 241

4.6.2 SMTP Client Implementation . . . 242

4.6.2.1 SMTP Client Data Structures . . . 242

4.6.2.2 SMTP Client Functions . . . 244

4.7 Queue Manager . . . 245

4.7.1 Queue Manager Implementation . . . 246

4.7.2 Locking . . . 246

4.7.2.1 Deadlock Avoidance . . . 247

4.7.3 Data Structures . . . 247

4.7.4 Data Flow . . . 251

4.7.5 Functions . . . 252

4.7.5.1 Initialization Functions . . . 252

4.7.5.2 Communication Functions . . . 252

4.7.5.3 Commit Task . . . 253

4.7.5.4 Scheduler . . . 253

4.7.5.5 QMGR to SMTPC Protocol . . . 254

4.7.5.6 Load Control Implementation . . . 254

4.7.5.7 Updating Recipient Status . . . 256

4.7.5.8 Handling Bounces . . . 260

4.7.5.9 DSN: Delayed . . . 261

4.7.5.10 AR to QMGR . . . 262

4.8 Address Resolver Implementation . . . 262

4.8.1 Resolving Recipient Addresses . . . 263

4.8.2 Alias Expansion . . . 264

4.8.3 Aliases: Owner- Handling . . . 264

4.9 Milter . . . 266

4.9.1 Sendmail 8 Milter Protocol . . . 266

4.9.1.1 Macros . . . 268
4.9.1.2 Option Negotiation . . . 268
4.9.2 SMTPS - Milter Protocol . . . 268
4.9.2.1 Option Negotiation . . . 270
4.9.2.2 Differences to Milter . . . 270
4.10 Implementation of Databases and Caches for Envelopes, Contents, and Connections . . . 271
4.10.1 Incoming Envelope Database . . . 271
4.10.2 Incoming Envelope Database Disk Backup . . . 271
4.10.3 Active Envelope Database . . . 271
4.10.4 Deferred Envelope Database Implementation . . . 275
4.11 Misc . . . 277
4.11.1 IBDB Cleanup . . . 278
4.11.2 Show Mail Queue . . . 278
4.12 Testing . . . 278
4.13 Problems encountered in the Implementation . . . 279
4.13.1 Implementation Problems . . . 279
4.13.1.1 Out of Memory . . . 279
4.13.1.2 Transaction Based Processing . . . 279
4.13.2 Behavior of the Implementation . . . 280
4.13.2.1 Number of Open Outgoing Connections . . . 280

5 Sendmail X: Performance Tests and Results 281

5.1 SMTP Server Daemon . . . 281
5.1.1 SMTP Sink . . . 281
5.1.1.1 SMTP Sink with CDB . . . 282
5.1.2 SMTP Relaying Using a Sendmail X Prototype . . . 284
5.1.2.1 Various Linux FS . . . 285
5.1.2.2 Various FreeBSD Results . . . 287
5.1.2.3 Various SunOS 5 Results . . . 292
5.1.2.4 Various OpenBSD Results . . . 292
5.1.2.5 Various AIX Results . . . 293
5.2 Implementation of Queues and Caches . . . 293
5.2.1 Filesystem Performance . . . 293
5.2.1.1 Test Systems . . . 293

5.2.1.2 Meta Data Operations 296
5.2.1.3 Writing a Logfile 298
5.2.2 Harddisk Performance 308
5.2.3 Performance of Berkeley DB 310
5.3 Miscellaneous about Performance 315
5.4 Performance of Various Programs 316
5.4.1 TCP/IP Performance 316
5.4.2 DB Lookup Performance 317
5.4.3 `snprintf` Performance 317

Chapter 1

Sendmail X: Requirements

This chapter describes the basic requirements which sendmail X.0 must fulfill as well as the basic functionality that it must implement. It will state several obvious items just for the sake of completeness. This chapter will serve as a basis from which the author will work to design sendmail X.0. The content must be agreed upon by all relevant, involved parties. It will also lay the groundwork for the future development of the sendmail X series of MTAs, of which sendmail X.0 will be the first implementation. sendmail X.0 is not meant to be feature complete but a proof of concept. It must be designed and written with extensibility in mind. Subsequent versions of sendmail X will add more features.

Note: some sections of this document are still under development.

1.1 Requirements

1.1.1 Basic Requirements

Some requirements for an MTA are so basic that it should not be necessary to mention them. However, we will do it nevertheless. These requirements are:

- Security: it must not be possible to abuse the MTA to compromise the security of the host on which the MTA is running.
- Reliability: the MTA *MUST NOT lose the message for frivolous reasons*[1].

These requirements are conditio sina qua non, they will be taken for granted throughout this and related documents and during the design and implementation of sendmail X. In addition to these, sendmail X must be efficient. However, neither of the two main requirements will be compromised to increase efficiency. The intended goal for the efficiency of sendmail X is to be about one order of magnitude faster than sendmail 8.

There is another obvious requirement which is given here for completeness, i.e., conformance to all relevant RFCs, esp. RFC 2821 [Kle01], and implementation of (nearly) all RFCs that sendmail 8 handles:

[1] RFC 2821, 6.1 [Kle01]

RFC 974	Mail Routing and the Domain System [Par86]
RFC 1123	Internet Host Requirements [Bra89]
RFC 1652	SMTP 8BITMIME Extension [KFR+94]
RFC 1869	SMTP Service Extensions [KFR+95]
RFC 1870	SMTP SIZE Extension [KFM95]
RFC 1891	SMTP Delivery Status Notifications [Moo96b]
RFC 1892	The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages [Vau96b]
RFC 1893	Enhanced Mail System Status Codes [Vau96a]
RFC 1894	Delivery Status Notifications [Moo96a]
RFC 1985	SMTP Service Extension for Remote Message Queue Starting [Win96]
RFC 2033	Local Mail Transfer Protocol [Mye96]
RFC 2034	SMTP Service Extension for Returning Enhanced Error Codes [Fre96]
RFC 2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies [FB96]
RFC 2476	Message Submission [GK98]
RFC 2487	SMTP Service Extension for Secure SMTP over TLS [Hof99]
RFC 2554	SMTP Service Extension for Authentication [Mye99]
RFC 2822	Internet Message Format [Res01]
RFC 2852	Deliver By SMTP Service Extension [New00]
RFC 2920	SMTP Service Extension for Command Pipelining [Fre00]

1.1.2 Other Requirements

sendmail X must be compliant with all the usual buzzwords (marketing or software development): robust, flexible, scalable, extendable, maintainable, portable to modern OS with support for POSIX (or similar) threads and certain other basic requirements. What this really means in particular will be explained in the next sections. sendmail X must run well on Unix and Windows (natively), support for Windows will not be added later on as an afterthought. It must provide hooks for monitoring, managing, etc. It also must be simple to replace certain modules (library functions) by specialized functions, e.g., as in sendmail 8.12 different lookup schemes for mailboxes.

sendmail X should be backward compatible with sendmail 8 at the user level as far as the basic requirements (esp. security) allow this. This includes support for aliases and `.forward` files, however the actual implementation may differ and result in slightly different behavior.

sendmail X should minimize the burden it puts on other systems, i.e., it must not overwhelm other MTAs, and it should make efficient use of network resources. It must properly deal with resource shortages on the local system and degrade gracefully in such events. However, the first implementation(s) of sendmail X may require some large amounts of certain resources, esp. memory.

sendmail X is meant to be useful to a majority of users, and thus must deal with some of the quirks of other MTAs and MUAs. However, it will most likely not go to the extreme of sendmail 8. Even though it should be "liberal in what it accepts", there are limits. Those limits are given by the time required to implement workarounds for broken systems, the possible problems for security (which will not be compromised), performance (which will only be compromised if really necessary), the amount of architectural changes, etc.

1.1.3 Explanation of Buzzwords

In this section we explain what the various buzzwords mean with respect to sendmail X.

1.1.3.1 Robustness

sendmail X must be robust which means that even in event of failures it should behave reasonably. It must never lose e-mails except in the case of hardware failures or software failures beyond sendmail's control that occur after sendmail has accepted responsibility for an e-mail (there's nothing sendmail can do if someone/something destroys the files/disks containing the e-mails). In the event of resource shortages (memory, disk, etc) it must degrade gracefully. It should be implemented in such a way that it can also deal with simple OS errors, e.g., if a (sub)process fails due to an error (e.g., looping because a system calls fails in unexpected ways), it should deal with that in a useful way, i.e., log an error message and exit without causing havoc to the rest of the system. It is not expected to deal with all possible errors (e.g., who supervises the supervisor?) but the design should be done properly to deal with most (also unexpected) problems.

sendmail X also must be able to deal with Denial of Service attacks as much as technically possible.

1.1.3.2 Flexibility

It must be possible to tune or replace components of the sendmail X system for different purposes to achieve good performance in almost all cases or to add new features.

1.1.3.3 Scalability

sendmail X should be able to take advantage of changes in the underlying OS and hardware that modify the relative performance of the available components. For example, if more processing power (faster or more CPUs) is available then the overall system should become faster. However, this can't be taken to an extreme since other parts might become bottlenecks despite careful design. For example, it is unreasonable to expect twice as many local mail deliveries if the number of CPUs is doubled since the limiting factor is most likely disk I/O in this scenario.

Clustering/High Availability support might be required too. The impact, as far as an application designer is concerned, is:

- the need to leave a consistent data set on disk after a crash;
- the need to start up quickly with a post-crash data set;
- the ability to play nicely with some kind of front-end load-balancer.

1.1.3.4 Extendability

It must be possible to extend sendmail X in various places. These include: interfaces to other mailbox identification schemes, interfaces to other map types, adding other delivery agents. sendmail X will have clear APIs for various extensions, as for example the milter API is one for sendmail 8.

1.1.3.5 Maintainability

sendmail X must be designed and implemented to be easily maintainable. This not just means adherence to the Sendmail coding standard, but also a clear design without "surprises", e.g., far reaching side effects.

1.1.3.5.1 Testing It is important that (almost) each functionality in sendmail X is subject to automatic testing. Having tests greatly simplifies maintainability because changes to the implementation can be automatically tested, i.e., if some of the tests break then the changes are most likely incorrect (unless the tests are broken themselves). Tests should cover (almost) all of the code (paths).

1.1.3.6 Portability

sendmail X must run on Unix OSs which support (POSIX) threads and on Windows. The exact list of requirements for the OS is not yet finalized. sendmail X will be written in C, most likely C89-compliant.

1.2 Restrictions

There are several things that sendmail X will not do or offer. First of all, it will not use the same (or even compatible) configuration files as sendmail 8 does. There should be a tool to migrate sendmail 8 `mc` files to sendmail X configuration files. However, it is unlikely that `cf` files can be easily migrated to sendmail X configuration files.

sendmail X.0 will not not have support for:

- MIME conversion (7/8 bit); it will just send 8 bit data (or maybe reject delivery based on some configuration option).
- UUCP support (other than external delivery agent). Support for other nets: bitnet, fido, DECNet, ... i.e., address parsing/syntax is RFC (2)821/(2)822.
- Special support for dialup systems.
- Localization.

Those features may be added in future versions.

1.3 Configuration

This section deals with two aspects: the configuration file for sendmail X and the configuration options for sendmail X.0.

1.3.1 Configuration File

sendmail 8 uses a configuration file that is supposedly easy to parse, but hard to read, understand, or modify by a system administrator. Even though this was simplified by using a macro preprocessor (`m4`) and a significantly simpler input file (`mc`), it is still considered complex. Moreover, there is no syntax check for the `mc` file which makes it fairly user unfriendly.

Hence it is necessary that sendmail X uses a simpler and more flexible configuration file. We need a syntax that allows us to keep options that belong together in a structured entity. This will be achieved with a C like syntax using (initial) keywords and braces. Moreover, the syntax will be free form, i.e., it will not depend on indentations, use of tabs, etc. It is also a bad idea to require special software, e.g., editors, to maintain the configuration files. For example, requiring a syntax oriented editor will add an additional migration hurdle which must be avoided otherwise users may switch to a different MTA that does not have such additional requirements.

Since sendmail X will consist of several modules, it is likely that there are several configuration files, at least for modules that are really separate. Moreover, it might be useful to delegate certain parts of a configuration to different administrators or even users. For example, it should be possible to delegate the configuration of virtual domains to different administrators. However, this will not be implemented in sendmail X.0, but the design will take this into consideration.

1.3.2 Configuration Options

sendmail X.0 will initially offer a subset of the (huge set of) configuration options of sendmail 8. It is simply not possible to implement all of the accumulated features of sendmail 8 in the first version of sendmail X within a reasonable timeframe and the limited resources we have.

1.4 Documentation

The documentation for sendmail X must fulfill different requirements with respect to its structure and its formats. The latter is explained in the next section.

The sendmail documentation must at least consist of the following sections:

- simple introduction: what is sendmail X, what does it do.
- common installations with examples: this will be the important part for wide-spread use of sendmail X. It should be a fast and short list of common examples.
- operational details: how does sendmail X work. This is usually required to understand in detail what sendmail X can do.
- hints for tuning: based on the operational details, how can sendmail X be tweaked. This may include hints for specific situations, e.g., high-volume mail server, limited memory, gateway functionality only.
- details (reference manual): the nitty-gritty details, containing each option and an explanation, including default value and acceptable values/ranges.

It is important that the documentation provides different levels of details. The current documentation is not particularly well structured or organized. It more or less requires to read everything, which, however, is not really necessary nor does it help someone who has a lot of other things to do besides installing an MTA. Even though the installation of an MTA usually requires a bit more than

```
./configure && make && make test && make install && start
```

it is not really necessary to read about (or even understand) all parts of the MTA.

1.4.1 Documentation Format

The documentation that describes sendmail X must be written in a format that allows easy generation of different versions including at least:

- plain ASCII text to read it online with a normal text editor or other simple tools like `more(1)`,
- a printable version for high quality output on a printer, e.g., PostScript or PDF,
- a hypertext version (HTML) to allow simple following of cross-references with an HTML browser.

Chapter 2

Sendmail X: Architecture

This chapter describes the architecture of sendmail X. It presents some possible design choices for various parts of sendmail X and explains why a particular choice has been made. Notice: several decisions haven't been made yet, there are currently a lot of open questions.

2.1 General Architecture of sendmail X

sendmail X consists of several communicating modules. A strict separation of functionality allows for a flexible, maintainable, and scalable program. It also enhances security by running only those parts with special privileges (e.g., `root`, which will be used as a synonym for the required privileges in this text) that really require it.

Some terms relevant for e-mail are explained in a glossary 2.17.

2.1.1 Overview

sendmail X consists of the following modules:

- Supervisor (MCP): responsible for starting all components and supervising them.
- Queue manager (QMGR): controls message routing through the MTA, mail delivery, and implements general policy.
- SMTP server (SMTPS): Accepts incoming mails.
- Address resolver (AR): performs address resolutions including various map lookups.
- Message Submission Program (MSP): Command line tool to transfer messages to the SMTP server.
- Delivery agents (DA): for local (LDA) and remote delivery, one specialized agent implements SMTP client side (SMTPC).

sendmail X uses persistent databases for content (CDB) and for envelope (routing) information (EDB). The content DB is written by the SMTP servers only, and read by the delivery agents. The envelope DBs are under complete control of the queue manager.

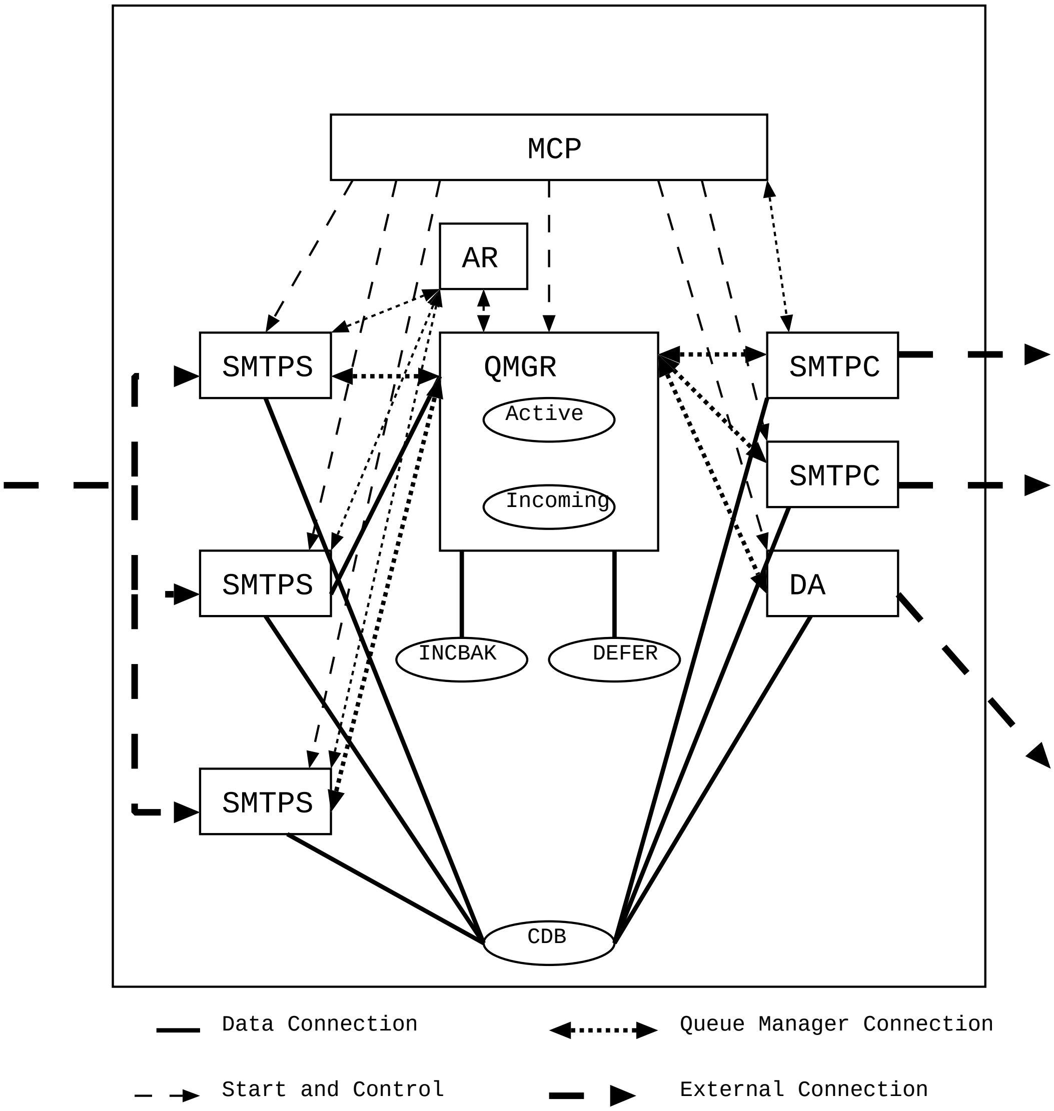

Figure 2.1: Sendmail X: Overall Structure

There are other components for sendmail X, e.g., a recovery program that can reconstruct an EDB after a crash if necessary, a program to show the content of the mail queue (EDB), and at least hooks for status monitoring.

2.1.2 Remarks about Performance

Since sendmail X is designed to have a lifetime of about one decade, it must not be tuned to specific bottlenecks in common computers as they are known now. For example, even though it seems common knowledge that disk I/O is the predominant bottleneck in MTAs, this isn't true in all cases. There is hardware support (e.g., disk system with non-volatile RAM) that eliminates this bottleneck[1]. Moreover, some system tests show that sendmail 8 is CPU bound on some platforms. Therefore the sendmail X design must be well-balanced and it must be easy to tune (or replace) subsystems that become bottlenecks in certain (hardware) configurations or situations.

2.1.3 Remarks about Configuration

This section contains some general remarks about configuring sendmail X. *Todo:* fill this in, add a new section later on that defines the configuration.

2.1.3.1 Simplicity and Flexibility

sendmail X must be easy enough to configure such that it does *not* require reading lots of files or even large section of a single file (see also Section 1.4). A "default" configuration may not require any configuration at all, i.e., the defaults should be stored in the binary and most of the required values should be automagically be determined at startup. A small configuration file might be necessary to override those defaults in case the system cannot determine the right values. Moreover, it is usually required to tell the MTS for which domain name to accept mail – by default a computer should have a FQDN but it is not advisable to decide to accept mail for the domain name itself[2]

The configuration file must be suitable for all kinds of administrators: at one end of the spectrum are those who just want to have an MTA installed and running with minimum effort, at the other end are those who want to tweak every detail of the system and maybe even enhance it by other software.

2.1.3.2 Option Grouping

Only a few configuration options apply globally, many have exceptions or suboptions that apply in specific situations. For example, sendmail 8 has timeouts for most SMTP commands and there are separate timeouts to return queued messages for different precedence values. Moreover, some features can be determined by rulesets, some options apply on a per connection basis, etc. In many cases it is useful to group configuration options together instead of having those options very fine grained. For examples, there are different SMTP mailers in sendmail 8 that create configuration groups (with some preselected set of options) which can be selected via mailertable (or rules). Instead of having mailer options per destination host (or other criteria), different options are grouped together and then an option set is selected. This can reduce the amount of configuration options that need to be stored (e.g., it's a two level mapping: address → mailer → mailer flags, instead of just one level in which each argument can have different function values: address → mailer and mailer flags).

However, it might be complicated to actually structure options in a tree like manner. For example, a rewrite configuration option may be

- per daemon, e.g., only applies to the MSA, not the MTA,
- per mailer, e.g., for local delivery but not for SMTP,
- per address type, e.g., for envelope recipient but not for header addresses.

[1] Price/MB for RAM approaches that for a hard disk

[2] For example, if the hostname is `host.domain.tld` then the system should by default accept mail for that name as local, but it should not treat `domain.tld` as local too.

Question: can we organize options into a tree structure? If not, how should we specify options and how should we implement them? Take the above example: there might be rewrite options per mailer and per address type (seems to make sense). However, in which order should those rewrite options be processed? Does that require yet another option?

A simple tree structure is not sufficient. For example, some option groups may share common suboptions, e.g., rewrite rules. Instead of having to specify them separately in each group, it makes more sense to refer to them. Here is an example from sendmail 8: there are several different SMTP mailers, but most of them share the same rewrite rulesets. In a strict tree structure each mailer would have a copy of the rewrite rulesets, which is neither efficient nor simple to maintain. Hence there must be something like "subroutines" which can be referenced. In a sendmail 8 configuration file this means there is a list of rulesets which can be referenced from various places, e.g., the binary (builtin ruleset numbers) and the mailers.

This means internally a configuration might be represented as a graph with references to various subconfigurations. However, this structure can be unfolded such that is actually looks like a tree. Hence, the configuration can conceptually be viewed as a tree.

2.1.3.3 Accessing and Changing the Configuration

There should be a way to query the system about the current configuration and to change (some) options on the fly. A possible interface could be similar to `sysctl(8)` in BSD. Here options are structured in a tree form with names consisting of categories and subscategories separated by dots, i.e., "Management Information Base" (MIB) style. Such names could be daemon.MTA.port, mailer.local.path, etc. If we can structure options into a tree as mentioned in the previous section then we can use this naming scheme. Whether it will be possible to change all parts on the fly is questionable, esp. since some changes must be done as transaction (all at once or none at all).

2.1.4 Security Considerations

Each section of this chapter that describes a module of sendmail X has a subsection about security considerations for that particular part. More discussion can be found in Section 2.14.

2.1.5 Control and Data Flow

This section gives an overview over the control and data flow for a typical situation, i.e., e-mail received via SMTP. This should give an idea how the various components interact. More details can be found in the appropriate sections.

1. An incoming message arrives via SMTP. The SMTP server (see Section 2.5) receives the connection attempt and contacts the queue manager (see Section 2.4) with that information. Both together decide whether to accept or reject the connection.

2. If SMTP commands are used that change the status of a session (e.g., STARTTLS, AUTH), those are executed and their effects are stored in the session context.

3. For each transaction a new envelope is created and the commands are communicated to the queue manager and the address resolver for validation and information. Other processes (esp. milters) might be involved too and the commands are either accepted or rejected based on the feedback from all involved processes.

4. When an e-mail is received (final dot), the queue manager and the SMTP server must either write the necessary information to stable (persistent) storage[3] or a delivery agent must take over and deliver the e-mail to all recipients immediately. The final dot is only acknowledged after either of these actions completed successfully.

5. The queue manager schedules delivery for the received e-mail in an appropriate matter (according to local policies and current system status etc). Note: delivery scheduling is done per recipient, not per envelope as it was in earlier sendmail versions. This makes it simpler to reuse open connections.

6. A delivery agent receives information from the queue manager which mail (content id) to send from which sender to which recipient(s). It informs the queue manager about success/(temporary) failure of its delivery attempts(s). The queue manager updates its EDB accordingly.

7. If all recipients in an envelope have been successfully delivered, the corresponding data is removed from the EDB (maybe triggering a DSN) and the content is removed from the CDB.

2.2 Configuration

2.2.1 General

Question: can we treat a configuration file like a programming language with

1. declarations: in which case do we need to declare things?

2. definitions: Definitions would be for things like SMTP servers: daemons which have attributes like port number and IP address to listen on, or delivery agents which have attributes like protocol and hostname to use.

3. functions (actions): These would be called by the binary in certain places (stages of the ESMTP protocol), e.g., when a connection is made, when a MAIL command is received etc.

Definitions do not depend on anything else, they define the basic structure (and behavior?) of the system. There are fixed attributes which cannot be changed at runtime, e.g., port number, IP address to listen on. Attributes which can change at runtime, e.g., the hostname to use for a session, fall in category 3, i.e., they are functions which can determine a value at runtime.

The distinction between definitions and functions is largely determined by the implementation and the underlying operating system as well as the application protocol to implement and the underlying transport protocol. When defining an SMTP daemon (or a DA) some of its attributes must be fixed (defined/specified) in the configuration, these are called *immutable*. For example, it is not possible to dynamically change the port of the SMTP daemon because that's the way the OS call `bind(2)`[4] works. However, the IP address of the daemon does not need to be fixed (within the capabilities of the OS and the available hardware), i.e., it could listen on exactly one IP address or on any. Such configuration options are called *variable* or *mutable*[5].

It seems to be useful to make a list of configuration options and their "configurability", i.e., whether they are fixed, or at which places they can change, i.e., on which other values they can depend.

[3]That is storage that does not lose its content during a system failure, e.g., power loss. Such storage is usually a disk, but not DRAM.

[4]It is possible not to specify a port but then the OS selects one which isn't useful for an SMTP server.

[5]Better name?

2.2.2 Syntax

As required, the semantics of the configuration file does not depend on its layout, i.e., spaces are only important for delimiting syntactic entities, tabs (whitespace) do not have a special meaning.

The syntax of the sendmail X configuration files is as follows:

<conf> ::= <entries>
<entries> ::= <entry> *
<entry> ::= <option> | <section>
<section> ::= <keyword> [<name>] ”{” <entries> ”}” [”;”]
<option> ::= <option-name> ”=” <value>
<value> ::= <name> ”;” | <values> [”;”]
<values> ::= ”{” <name-list> ”}”

This can be shortened to (remove the rule for entries):

<conf> ::= <entry> *
<entry> ::= <option> | <section>
<section> ::= <keyword> [<name>] ”{” <conf> ”}” [”;”]
<option> ::= <option-name> ”=” <value>
<value> ::= <name> ”;” | <values> [”;”]
<values> ::= ”{” <name-list> ”}”

Generic definition of “list”:

<X-list> ::= <X> | <X> ”,” <X-list> [”,”]

That is, a configuration file consists of a several entries, each of which is either a section or an option. A section starts with a keyword, e.g., mailer, daemon, rewriterules, and has an optional name, e.g., daemon MTA. Each section contains a section of entries which is embedded in curly braces. Each syntactic entity that isn't embedded in braces is terminated with a semicolon. An entry in a section can be an option or a (sub)configuration. To make writing configuration files simpler, lists can have a terminating comma and a semicolon can follow after <values>. That makes these symbols terminators not separators.

Examples:

```
mailer smtp {
    Protocol = SMTP;
    Connection = TCP;
    Port = mtp;
    flags { DSN }
    MaxRecipientsPerSession = 5;
};
mailer lmtp {
    Protocol = LMTP;
    flags = { LocalRecipient, Aliases }
    Path = "/usr/bin/lmtp";
};
Daemon MTA {
    smtps-restriction = { qualified-sender, resolvable-domain }
};
Map Mailertable { type = hash; file = "/etc/smx/mailertable"; };
Rewrite {
```

```
    Envelope {
        sender = { Normalize, Canonify },
        recipient = { Normalize, Virtual, Mailertable }
    };
    Header {
        sender = { Normalize },
        recipient = { Normalize }
    };
};
Check {
    DNSBL MyDNSBL { Orbd, Maps }
    Envelope {
        sender = { Qualified, MyDNSBL },
        recipient = { Qualified, AuthorizedRelay }
    };
};
```

The usual rules for identifiers (list of characters, digits, and underscores) apply. Values (<name>) that contain spaces must be quoted, other entries can be quoted, but don't need to. Those quotes are stripped in the internal representation. Backslashes can be used to escape meta-symbols.

Todo: completely specify syntax.

Note: it has been proposed to make the equal sign optional for this rule:

<option> ::= <option-name> ["="] <value>

However, that causes a reduce/reduce conflict when the grammar is fed into `yacc(1)`[6] because it conflicts with

<section> ::= <keyword> [<name>] "{" <entries> "}" [";"]

That is, with a lookahead of one it can not be decided whether something reduces to <option> or <section>. If the parser "knows" whether some identifier is a keyword or the name of an option then the equal sign can easily be optional. However, doing so violates the layering principle because it "pushes" knowledge about the actual configuration file into the parser where it does not really belong: the parser should only know about the grammar. Of course if would be possible to write a more specific grammar that includes lists of options and keywords. However, keeping the grammar abstract (hopefully) allows for simpler tools to handle configuration files. Moreover, if new options or keywords are added the parser does not need to change, it is only the upper layers that perform semantic analysis of a configuration file.

2.2.2.1 Comments

Most configuration/programming languages provide at least one way to add comments: a special character starts a comment which extends to the end of the line. Some languages also have constructs to end comments at a different place than the end of a line, i.e., they have characters (or character sequences) that start and end a comment. To make it even more complicated, some languages allow for nested comments. Text editors make it fairly easy to replace the begin of a line with a character and hence it is simple to "comment out" entire sections of a (configuration) file. Therefore it seems sufficient to have just a simple comment character ("#") which starts a comment that extends to the end of the current line. The comment character can be escaped, i.e., its special meaning disabled, by putting a backslash

[6] Thanks to John Kennedy for testing this.

in front of it as usual in many languages.

2.2.2.2 Internationalization

For now all characters are in UTF-8 format which has ASCII has a proper subset. Hence it is possible to specify texts in a different language, which might be useful in some cases, esp. if the configuration syntax is also used in other projects than sendmail X.

2.2.2.3 Strings and other Types

Strings are embedded (as usual) in double quotes. To escape special characters inside strings the usual C conventions are used, probably enhanced by a way to specify unicode characters ("\uVALUE"). Strings can not continue past the end of a line, to specify longer strings they can be continued by starting the next line (after any amount of white space) with a double quote (just like in ANSI C).

The parser should be able to to some basic semantic checks for various types. That is, it can detect whether strings are well formed (see above), and it must understand basic types like boolean, time specification, file names, etc.

2.2.2.4 Inclusion

There has been a wish to include configuration data via files or even databases, e.g., OpenLDAP attributes.

2.2.3 Alternative Configuration Formats

There are some suggestions for alternative configuration formats:

1. a simple style that uses a line oriented syntax:

   option = value

   This syntax is not flexible enough to describe the configuration of an MTA, unless some hacks are employed as done by postfix which uses an artificial structuring by naming the options "hierarchically". For example, sendmail 8 uses a dot-notation to structure some options, e.g., timeouts (`Timeout.queuereturn.urgent`); postfix uses underscores for a similar purpose, e.g.,

   ```
   smtpd_recipient_restrictions =
   smtpd_sender_restrictions =

   local_destination_concurrency_limit =
   default_destination_concurrency_limit =
   ```

   An explicit hierarchical structure is easier to understand and to maintain.

2. XML. While this is a syntax that is easily parsable by a computer, it is hard to read and especially hard to edit by a human. According to the requirements stated in Section 1.3.1, the latter is a problem that must be avoided.

The syntax of a configuration file could be easily changed to fit various *tastes*. For example, instead of using braces and section a "flat" format could be used that is semantically equivalent. This part of the example listed earlier:

```
Rewrite {
    Envelope {
        sender = { Normalize, Canonify },
        recipient = { Normalize, Virtual, Mailertable }
    };
};
Check {
    DNSBL MyDNSBL { Orbd, Maps }
    Envelope {
        sender = { Qualified, MyDNSBL },
        recipient = { Qualified, AuthorizedRelay }
    };
};
```

could be also written as:

```
Rewrite.Envelope.sender= Normalize, Canonify
Rewrite.Envelope.recipient = Normalize, Virtual, Mailertable
Check.DNSBL[MyDNSBL]= Orbd, Maps
Check.Envelope.sender = Qualified, MyDNSBL
Check.Envelope.recipient = Qualified, AuthorizedRelay
```

Of course other delimiters than dot (for hierachy) and brackets (for section names) can be used. However, it is easy to define a bijective function that transforms configuration files from one syntax into another and preserving the semantics. This also applies for transformation to/from XML. There could be various frontends that scan and parse a configuration file according to some syntax and generate the same internal structure that is then used by some applications.

2.2.4 Configurability

SMTP defines a structure which influences how a SMTP server (and client) can be configured. The topmost element in SMTP is a session, which can contain multiple transactions, which can contain multiple recipients and one messsage. Each of these elements has certain attributes (properties). For example

- a SMTP session has:
  - a client IP address and port, a server IP address and port (the IP quadruple).
  - possibly STARTTLS and AUTH active, including their attributes, e.g., TLS version, SASL mechanism, cipher algorithm, and key length.
- a SMTP transaction has:
  - a sender address and optional arguments, e.g., for DSN.
- a SMTP recipient has:

– an address and optional arguments.

This structure restricts how a SMTP server can be configured. Some things can only be selected ("configured") at a certain point in a session, e.g., a milter can not be selected for each recipient[7], neither can a server IP address selected per transaction, other options have explicit relations to the stage in a session, e.g., MaxRecipientsPerSession, MaxRecipientsPerTransaction (which might be better expressed as Session.MaxRecipients and Transaction.MaxRecipients or Session.Transaction.MaxRecipients). Some options do not have a clear place in a session at all, e.g., QueueLA, RefuseLA: do these apply to a session, a transaction or a recipient? It is possible to use QueueLA per recipient, but only in sendmail X because it does scheduling per recipient, in sendmail 8 scheduling is done per transaction and hence QueueLA can only be per transaction. This example shows that an actual implementation restricts the configurability, not just the protocol itself.

If a SMTP session is depicted as a tree (where the root is a session) then there is a "maximum depth" for each option at which it can be applied. As explained before, that depth is determined

1. by the structure of SMTP,
2. explicitly by tying it to an SMTP stage,
3. by the implementation.

Question: taking these restrictions into consideration, can we specify the maximum depth for each configuration option at which the setting the option is possible/makes sense? Moreover, can we specify a range of depths for options? For example: QueueLA can be a global option, an option per daemon, an option per session, etc. If such a range can be defined per option, then the configuration can be checked for violations. Moreover, it restricts the "search" for the value of an option that must be applied in the current stage/situation.

Question: it seems the most important restriction is the implementation (beside the structure of SMTP of course). If the implementation does not check for an option at a certain stage, then it does not make any sense to specify the option at that stage. While for some options it is not much effort to check it at a very deep level, for others that means that data structures must be replicated or be made significantly more complex. Examples:

1. Checking a simple integer value against a global value, e.g., RefuseLA or MinFreeDiskSpace, at the recipient stage is fairly easy, it requires just a few lines of code at that place. Checking RefuseLA at session, transaction, and recipient stage requires duplication of code, at least some function must be called at all of the places.
2. Checking the connection rate per recipient[8] requires that the structure which stores the connection rate actually takes the recipient into account instead of just the client IP address. This obviously makes the structure more complicated, in this case the implementation would need to be changed significantly. Another example: reject mail from an IP address unless the recipient is postmaster. Here the application of the option must be "delayed" until all required information (IP address and recipient) is available. More about this... can this specified in the configuration? Per recipient rejection:

```
recipient postmaster { reject-client-ip {map really-bad-abusers} }
recipient * { reject-client-ip {map all-abusers} }
```

[7]this is a consequence of the way milters are designed, not a direct consequence of SMTP.
[8]this may not be a particular good example.

This brings us back to the previous question: *Question*: can we specify the maximum depth for each configuration option at which the setting the option makes sense or at which it is possible without making the implementation too complex.

There are other configuration options which do not really belong to that structure, e.g., "mailers" (as they are called in sm8). A mailer defines a delivery agent (DA), it is selected per recipient. Hence a DA describes the behavior of an SMTP client, not an SMTP server. In turn, many options are per DA too, while others only apply to the server, e.g., milters are server side only[9].

Problem: STARTTLS is a session atttribute, i.e., whether it is used/offered is defined per client/server (per session). However, it is useful (and possible) to require certain STARTTLS features per recipient[10] (as sm8 does via access db and ruleset). It is not possible to say: only offer STARTTLS feature X if the recipient is R, but it is possible to say: if the recipient is R then STARTTLS feature X must be in use (active). Moreover, it's not possible to say: "if the recipient is R, the milter M must be used." How do those configuration options fit into the schema explained above? What's the qualitative difference between these configuration options?

1. STARTTLS is offered by the server and selected by the client, there is a negotiation in the handshake. If TLS feature X is required for recipient R then there is usually and out-of-band agreement between MTA admins to do that, because this cannot be negotiated in ESMTP[11].
2. Milter is not part of SMTP, it is solely defined by the local admin and by the design/implementation. Changing the features of a milter based on a recipient cannot be done in ESMTP but in the milter itself.
3. STARTTLS features can be set per recipient in SMTPC. Does that make STARTTLS an option per mailer, i.e., a mailer defines whether it uses STARTTLS (including its attributes)? The definition of a DA is probably similar in structure to the definition of a SMTP server. However, it is possible to specify session behavior per recipient, because a DA is selected per recipient[12].

Questions: What's the qualitative difference between these examples? What is the underlying structure? How does the structure define configurability, i.e., what defines why a behavior/option can be dependent on something but not on something else?

For example: STARTTLS in client (SMTPC): this isn't really: "use STARTTLS with feature X if recipient R will be send", but it is: "if recipient R will be send then STARTTLS with feature X must be active" (similar to SMTPS). However, it is conceivable to actually do the former, i.e., make a session option based on recipient because smX can do per recipient scheduling, i.e., a DA is selected per recipient. Hence it can be specified that a session to deliver recipient R must have STARTTLS feature X. However, doing that makes connection reuse significantly more complicated (see Section 3.4.10.2). *Question*: doesn't this define a specific DA? Each DA has some features/options. Currently the use of STARTTLS is orthogonal to DAs (e.g., almost completely independent) hence the connection reuse problem (a connection is defined by DA and server, not DA and server and specific features because those features should be in the DA definition). Hence if different DAs are defined based on whether STARTTLS feature X should be used, then we tied a session to DA and server. This brings us to the topic of defining DAs. *Question*: what do we need "per DA" to make things like connection reuse simple? Note: if we define DAs with all features, then we may have a lot of DAs. Hence we should restrict the DA features to those which are really specific to a DA (connection/session/transaction) behavior, and cannot be defined independently. For example,

[9]at least right now.
[10]that is, rejecting a recipient if the feature has not been used/is not active.
[11]Because STARTTLS is per session, not per recipient, hence it is not possible to do this in the server, but it is (theoretically) possible in the client.
[12]which is an implementation decision, with a different implementation this would not be possible.

it doesn't seem to be useful to have a DA for each different STARTTLS and AUTH feature, e.g., TLS version, SASL mechanism, cipher algorithm, and key length. However, can't we leave that decision up to the admin?

2.2.4.1 Consistency

In addition to simple syntax checks, it would be nice to check a configuration also for consistency. Examples?

2.2.5 Configuration File Structure

As explained in Section 2.1.3.2 there are some issues with the structuring of the configuration options. Here is a simple example that should serve as base for a discussion:

```
Daemon MTA {
 smtps-restriction { qualified-sender, resolvable-domain }
 mailer smtp { Protocol SMTP; Port smtp; flags { DSN }
    MaxRecipientsPerSession 25;
 };
 Aliases { type hash; file /etc/smx/aliases; };
 mailer lmtp { Protocol LMTP; flags { LocalRecipient, Aliases }
    Path "/usr/bin/lmtp";
 };
 Map Mailertable { type hash; file /etc/smx/mailertable; };
 Rewrite {
    Envelope { sender { Normalize },
               recipient { Normalize, Virtual, Mailertable }
    };
    Header { sender { Normalize }, recipient { Normalize } };
 };
};

Daemon MSA {
 mailer smtp { Protocol SMTP; Port submission; flags { DSN }
    MaxRecipientsPerSession 100;
 };
 Aliases { type hash; file /etc/smx/aliases; };
 mailer lmtp { Protocol LMTP; flags { LocalRecipient, Aliases }
    Path "/usr/bin/lmtp";
 };
 Rewrite {
    Envelope { sender { Normalize, Canonify },
               recipient { Normalize, Canonify }
    };
    Header { sender { Normalize, Canonify  },
             recipient { Normalize, Canonify  } };
 };
};
```

This configuration specifies two daemons: MTA and MSA which share several subconfigurations, e.g., aliases and lmtp mailer, that are identical in both daemons. As explained in Section 2.1.3.2 it is better to not duplicate those specifications in various places. Here is the example again written in the new style:

```
aliases MyAliases { type hash; file /etc/smx/aliases; };
mailer lmtp { Protocol LMTP; flags { LocalRecipient, Aliases }
   Path "/usr/bin/lmtp";
};

Daemon MTA {
 smtps-restriction { qualified-sender, resolvable-domain }
 mailer smtp { Protocol SMTP; Port smtp; flags { DSN }
    MaxRecipientsPerSession 25;
 };
 aliases MyAliases;
 mailer lmtp;
 Map Mailertable { type hash; file /etc/smx/mailertable; };
 Rewrite {
    Envelope { sender { Normalize },
               recipient { Normalize, Virtual, Mailertable }
    };
    Header { sender { Normalize }, recipient { Normalize } };
 };
};

Daemon MSA {
 mailer smtp { Protocol SMTP; Port submission; flags { DSN }
    MaxRecipientsPerSession 100;
 };
 aliases MyAliases;
 mailer lmtp;
 Rewrite {
    Envelope { sender { Normalize, Canonify },
               recipient { Normalize, Canonify }
    };
    Header { sender { Normalize, Canonify  },
             recipient { Normalize, Canonify  } };
 };
};
```

Here the subconfigurations aliases and lmtp mailer are referenced explicitly from both daemon declarations. This is ok if there are only a few places in which a few common subconfiguration are referenced, but what if there are many subconfigurations or many places? In this case a new root of the tree would be used which declares all "global" options which can be overridden in subtrees. So the configuration tree would look like:

```
generic declarations
common root
  daemon
    mailer
      ?
```

Question: what is the complete structure of the configuration tree? *Question*: can the tree be specified by the configuration file itself, or is its structure fixed in the binary?

The next problem is how to find the correct value for an option. For example, how to determine the value for `MaxRecipientsPerSession` in this configuration:

```
MaxRecipientsPerSession 10;
Daemon MTA {
 MaxRecipientsPerSession 25;
 mailer smtp { ... }; };
 mailer relay { MaxRecipientsPerSession 75; }; };
Daemon MSA {
 MaxRecipientsPerSession 50;
 mailer smtp { MaxRecipientsPerSession 100; }; };
```

Does this mean the system has to search in the tree for the correct value? This wouldn't be particularly efficient.

sendmail 8 also offers configuration via the `access` database, i.e., some tagged key is looked up to find potential changes for the configuration options that are specified in the `cf` file. For example, `srv_features` allows to set several options based on the connecting client (see also Section 2.2.7.2). This adds another "search path" to find the correct value for a configuration option. In this case there are even two tree structures that need to be searched which are defined by the host name of the client and its IP address, both of which are searched for in the database by removing the most significant parts of it, e.g., `Tag:host.sub.tld`, `Tag:sub.tld`, `Tag:tld`, `Tag:`.

2.2.6 Option Names

Option names should be meaningful and intuitive, i.e., a name should convey what the option does or specifies, and when someone wants the system to do something, he should find the corresponding option name fairly easily. Example: `fd_socket` is not a good option name, it is not clear what the option does; `socket_to_pass_fd` is significantly better.

All option names should be consistent, i.e., if one name uses a certain term for some item then another option must not use a different term for the same item. This also include the spelling of option names as well as their structure. Example: do not use `path` and `filename` to denote the same thing.

See also Section 3.2.2.

2.2.7 Dynamic Configuration

What about a "dynamic" configuration, i.e., something that contains conditions etc? For example:

```
if client IP = A and LA < B
then
  accept connection
else if client IP in net C and LA < D and OpenConnections < E
then
  accept connection
else if OpenConnections < F
```

```
then
  accept connection
else if ConnectionRate < G
then
  accept connection
else
  reject connection
fi
```

Note: it might be not too hard to specify a functional configuration language, i.e., one without side effects. However, experience with sm8 shows that temporary storage is required too[13]. As soon as assignments are introduced, the language becomes significantly more complex to implement. Moreover, having such a language introduces another barrier to the configuration: unless it is one that is established and widely used, people would have to learn it to use smX efficiently. For example, the configuration language of exim allows for runtime evaluation of macros (variables) and the syntax is hard to read (as usual for unknown languages). There are a few approaches to deal with this problem:

1. define clean interfaces for C, i.e., function calls, their API, how variables can be accessed, etc. This is something like the module concept discussed elsewhere (2.13 and 3.17).

2. provide an interface to external languages, e.g., perl, python, etc. This would be very flexible but makes implementation complicated.

2.2.7.1 Configuration: Conditionals

One proposal for the smX syntax includes conditionals in the form of

<entry> ::= <option> | <section>
<option> ::= <option-name> [”=”] <value>
<condopt> ::= ”if” ”(” <condition> ”) <option>

2.2.7.2 Configuration: Reference to Maps

In sendmail 8 it proved to be useful to have some configuration options stored in maps. These can be as simple as reply codes to certain phases in ESMTP and for anti-relay/anti-spam checks, and as complex as the `srv_features` rulesets (see also Section 2.2.5).

There are several reasons to have configuration options in maps:

1. Maps provide a efficient lookup features (normal maps however do not offer any form of pattern matching). This is important for things like blacklists which in some cases contains thousands if not hundreds of thousands of entries.

2. In sendmail 8 changing a configuration file requires restarting the daemon, while changing maps can be done without that interruption (sm8 dynamically opens maps).

If not just anti-spam data is stored in maps but also more complicated options (as explained before: map entries for `srv_features`) then those options are usually not well structured, e.g., for the example it is

[13] See `storage` map in sm8.

just a sequence of single characters where the case (upper/lower) determines whether some features is offered/required. This does not fulfill the readability requirements of a configuration syntax for smX.

Question: how to integrate references to maps that provide configuration data into the configuration file syntax and how should map entries look like? One possibility way is to have a set of option combined into a group and reference that group from the map. For example, instead of using

```
SrvFeatures:10          l V
```

it would be

```
LocalSrvFeatures { RequestClientCertificate=No; AUTH=require; };

SrvFeatures:10         LocalSrvFeatures
```

2.2.8 Default Configuration

The defaults of the configuration should be compiled into the binary instead of having a required configuration file which contains all default values.

Advantages:

1. The default configuration file would be rather larger if it needs to have all default settings for all configuration options.
2. there's no need to have a big configuration file that sets all the necessary options (even if that "default" configuration file is just "included" somehow in the "user" configuration file)
3. avoids out of sync between a default configuration file and the binary: new options have the correct defaults, "useful" changes apply without updating the default configuration file.
4. people will edit the default configuration file, in sm8 this has been noticed for `devtools/OS/file`, `sendmail.cf`, and other examples, e.g,, startup files in an OS, and hence make upgrading hard.

Disadvantages:

1. this may change "silently" options, and even though it's documented in the release notes, people will still be "surprised".

2.2.9 Displaying Current Configuration

It must be possible to query the various smX components to print their current configuration settings as well as their current status. The output should be formatted such that it can be used as a configuration file to reproduce the current configuration.

2.2.10 Dynamically Changing Configuration

It must be possible to tell the various smX components to change their current configuration settings. This may not be practical for all possible options, but at least most of them should be changeable while the system is running. That minimizes downtime to make configuration changes, i.e., it must not be required to restart the system just to change some "minor" options. However, options like the size of various data structures may not be changeable "on the fly".

2.2.11 Configuration Example for PreAlpha Version

sendmail X.0.0.PreaAlpha9 has the following configuration parameters:

1. QMGR:

   (a) definitions (see Section 2.2.1, 2): size of various data structures: AQ, IBDB, IQDB rsc size, IQDB hash table size.

   Various other definitions: postmaster address for double bounces[14], log level and debug level could be more specific, i.e., per module in the code, but probably not per something external, configuration flags, time to wait for SMAR, SMTPC to be ready.

   It doesn't seem to be very useful to make these dependent on something: minimum and "ok" free disk space (KB).

   (b) initial and maximum number of concurrent connections to one IP addr: this could be dependent on DA and destination IP address.

   (c) initial and maximum delay for retries, maximum time in queue: this could be dependent on many things, e.g., precedence, size, sender, recipient.

   (d) maximum connection rate per 60s, maximum number of open connections (SMTPS): these could be dependent on SMTP server and client IP address.

   (e) maximum time before scheduling a DSN, maximum time in AR and in DA.

2. SMTPS:

   definitions (see Section 2.2.1, 2): log level and debug level (see above), heap check level, group id (numeric) for CDB, time to wait for QMGR to be ready.

   run in interactive mode, serialize all accept() calls, perform one SMTP session over stdin/stdout,

   socket over which to receive listen fd, specify thread limits per listening address,

   create specified number of processes, bind to specified address – multiple addresses are permitted, maximum length of pending connections queue,

   I/O timeout: could be per daemon and client IP address,

   client IP addresses from which relaying is allowed, recipient addresses to which relaying is allowed.

3. SMTPC:

   All of these are definitions:

   log level and debug level (see above), heap check level, time to wait for QMGR to be ready, run in interactive mode, create specified number of processes, specify thread limits.

   These could be dependent on DA or even server address: socket location for LMTP, I/O timeout, connect to (server)port.

[14]this could be per domain, but which domain?

4. SMAR:

   All of these are runtime options, i.e., they are specified when the binary is started (hence definitions in the sense of Section 2.2.1, 2):

   log level and debug level (see above), IPv4 address for name server, DNS query timeout, use TCP for DNS queries instead of UDP, use connect(2) for UDP.

5. MCP:

   All of these are definitions: name: string (name of program/service); port: number or service entry (optional); socket.name: name of socket to listen on: path (optional); tcp: currently always tcp (could be udp); type: type of operation: nostartaccept, pass, wait; exchange_socket: socket over which fd should be passed to program; processes_min: minimum number of processes; processes_max: maximum number of processes; user: run as which user (user name, i.e., string); path: path to executable; args: arguments for `execv(3)` call.

```
MCP {
  processes_min=1; processes_max=1; type=wait;
  smtps {
    port=25;
    type=pass;
    exchange_socket=smtps/smtpsfd;
    user=smxs;
    path="../smtps/smtps";
    arguments="smtps -w 4 -d 4 -v 12 -g 262 -i -l . -L smtps/smtpsfd"; }
  smtpc {
    user=smxc;
    path="../smtpc/smtpc";
    arguments="smtpc -w 4 -P 25 -d 4 -v 12 -i -l ."; }
  qmgr {
    user=smxq;
    path="../qmgr/qmgr";
    arguments="qmgr -w 4 -W 4 -B 256 -A 512 -d 5 -v 12"; }
  smar {
    user=smxm;
    path="../smar/smar";
    arguments="smar -i 127.0.0.1 -d 3 -v 12"; }
  lmtp {
    socket_name="lmtpsock";
    socket_perm="007";
    socket_owner="root:smxc";
    type=nostartaccept;
    processes_min=0;
    processes_max=8;
    user=root;
    path="/usr/local/bin/procmail";
    arguments="procmail -z"; }
};
```

Note: some definitions could be functions (see Section 2.2.1), e.g., I/O timeout could be dependent on the IP address of the other side or the protocol, debug and log level could have similar dependencies. As explained in Section 2.2.4 the implementation restricts how “flexible” those values are.

Currently hostname is determined by the binary at runtime. If it is set by the configuration then it could be: global, per SMTP server, per IP address of client, per SMTP client, per IP address of server. This is one example of how an options can be set at various depths in the configuration file. Would this be a working configuration file?

```
Global { hostname = my.host.tld; }
Daemon SMTPS1 { Port=MTA;  hostname=my2.host.tld;
  IP-Client { IP-ClientAddr=127.*; hostname=local.host.tld;} }
DA SMTPC1 { hostname=out1.host.tld;
  IP-Server { IP-ServerAddr=127.*; hostname=local.host.tld;}
  IP-Server { IP-ServerAddr=10.*; hostname=net.host.tld;} }
```

The lines that list an IP address are intended to act as restrictions, i.e., if the IP address is as follows then apply this setting. *Question*: Is this the correct way to express that? What about more complicated expressions (see Section 2.2.7)?

In principle these are conditionals:

```
hostname = my.host.tld;
if (Port==MTA) { hostname=my2.host.tld;
  if (IP-ClientAddr==127.*) hostname=local.host.tld; }
```

2.2.11.1 Anti-Spam Configuration for PreAlpha Version

Question: what are the requirements for anti-spam configuration for a (pre-)alpha version of sendmail X?

1. allow relaying from a client IP address (client host name is not yet supported: no reverse lookup), or to a recipient domain. This is currently implemented via regular expressions, it would be nice to extend this to map lookups.

   Not yet available: allow relaying based on TLS.

2. rejection of client IP address, sender address (parts of it), recipient address.

   This brings in all the subtleties from sm8, especially delay-checks. What's a simple way to express this?

   The Control flow in sm8 is explained in Section 3.5.2.6.

Note: for the first version it seems to the best to use a simple configuration file without any conditionals etc. If an option is dependent on some data, then the access method from sm8 should be used. This allows us to put that data into a well known place and treat it in a matter that has been successfully used before. Configuration like anti-relaying should be "hard-wired" in the binary and their behavior should only be dependent on data in a map. This is similar to the mc configuration "level" in sm8; more control over the behavior is archievable in sm8 by writing rules which in smX may have some equivalent in modules.

2.2.12 Security Considerations

The configuration files must be protected from tampering. They should be owned by root or a trusted user. sendmail must not read/use configuration files from untrusted sources, which not just means wrong owners, but also files in unsecure directories.

2.3 Supervisor

Some processes require root privileges to perform certain operations. Since sendmail X will not have any set-user-id root program for security reasons, those processes must be started by root. It is the task of the *supervisor* process (MCP: Master Control Process) to do this.

There are a few operations that usually require root privileges in a mail system:

1. Bind to port 25 on Unix.
2. Delivery to a mailbox (file) that is owned by the recipient. [15]
3. In some cases access to restricted data might be necessary, e.g., root-only maps.

The MCP will bind to port 25 and other ports if necessary before it starts the SMTP server daemons (see Section 2.5) such that those processes can use the sockets without requiring root access themselves.

The supervisor process will also be responsible for starting the various processes belonging to sendmail X and *supervising* them, i.e., deal with failures, e.g., crashes, by either restarting the failed processes or just reporting those crashes if they are fatal. The latter may happen if a system has a hardware or software failure or is (very) misconfigured. The MCP is also responsible for shutting down the entire sendmail X system on request.

The configuration file for the supervisor specifies which processes to start under which user/group IDs. It also controls the behavior in case of problems, i.e., whether they should be restarted, etc. This is fairly similar to inetd, except that the processes are not started on demand (incoming connection) but at startup and whenever a restart is necessary.

2.3.1 Security Considerations

The supervisor process runs as root and hence must be carefully written (just like any other sendmail X program). Input from other sources must be carefully examined for possible security implications (configuration file, communication with other parts of the sendmail X system).

2.4 Queue Manager

The queue manager is the central coordination instance in sendmail X. It controls the flow of e-mail throughout the system. It implements (almost) all policies and coordinates the receiving and sending processes. Since it controls several processes it is important that it will not slow them down. Hence the queue manager will be a multi-threaded program to allow for easy scalability and fast response to requests.

[15] There are systems in which local delivery doesn't need to be done as the recipient, because the recipient is not a user of the OS. In that case access must be authorized by other means which however is beyond the scope of sendmail (and this document). Other systems, e.g., System V based Unix versions, use a group-writable mailbox which does not require root access either.

The queue manager will handle several queues (see Section 2.4.1); there will be at least queues for incoming mails, for scheduling delivery of mails, and for storing information about delayed mails.

The queue manager also maintains the state of the various other processes with which it communicates and which it controls, e.g., the open connections of the delivery agents. It should also have knowledge about the system to which it sends e-mails, i.e., whether they are accepting e-mails, probably the throughput of the connections, etc.

Todo: add a complete specification what the QMGR does; at least the parts that aren't related to incoming SMTP.

2.4.1 Queues

One proposal for a set of possible queues is:

- incoming: queue (memory) in which currently incoming envelope data is stored, backed up on disk (INCEDB).
- active: queue (memory) from which deliveries are scheduled (AQ or ACTEDB).
- deferred: queue (disk) in which envelope data is stored which refers to recipients that could have not been delivered due to temporary problems (DEFEDB). This queue is also called "delayed" queue.
- bounce: queue (disk) to store information about permanently failed delivery attempts. This queue can be used to coalesce DSNs, see also Section 2.4.6. Question: should it also used be to store data for other DSNs, i.e., success and warning?
- hold: queue (disk) in which envelope data is stored due to policy decisions.
- ETRN: queue (disk) which stores envelopes that are only scheduled on ETRN requests.
- corrupt: queue (disk) which holds envelope data that got corrupted somehow; for further inspection by a clueful person.

Having several on-disk queues has the following advantages:

- fewer entries per queue may result in faster search/access. It probably increases performance because items that are on hold/ETRN must be not tried, they are skipped automatically during queue lookups (to get a new entry into the active queue).
- special-purpose queues may lead to special-purpose (faster) implementations.
- it is possible to use different disks for different queues, e.g., slow disks for hold/ETRN, very fast disks for incoming.

Disadvantages of several on-disk queues are:

- several different APIs: more work, more potential for errors.
- migration between queues is more complicated and possibly slower.
- specialized queues for every purpose may make it harder to add new behavior. sendmail 8.12 showed that tying queues to behavior isn't a good idea.

- fragmentation (the system has spare disk space available, but can't proceed because one queue's partition has filled).

Since the disadvantages outweigh the advantages the number of on-disk queues will be minimized. The deferred queue will become the main queue and contains also entries that are on hold or waiting for ETRN. Envelopes for bounces should go to the main queue too. This way the status for an envelope is available in one place (well, almost: the incoming queue may have the current status). Only the "corrupt" queue is different since nobody ever schedules entries from this queue and it probably needs a different format (no decision yet). To achieve the "effect" of having different queues it might be sufficient to build different indices to access parts of the queue (logical queues). For example, the ETRN queue index has only references to items in the queue that are waiting for an ETRN command.

The "active" and the "incoming" queues are resident in memory for fast access. The incoming queue is backed up on stable storage in a form that allows fast modifications, but may be slow to reconstruct in case the queue manager crashes. The active queue is not directly backed up on disk, other queues act as backup. That is, the active queue is a restricted size cache (RSC) of entries in other queues. The deferred queue contains items that have been removed from the active queue for various reasons, e.g., policy (deliver only on ETRN, certain times, quarantine due to milter feedback, etc), delays (temporary failures, load too high, etc), or as the result of delivery attempts (success/failure/delay). The active queue must contain the necessary information to schedule deliveries in efficient (and policy based) ways. This implies that there is not just one way to access the data (one key), but several to accommodate different needs. For example, it might be useful to MX-piggyback deliveries, which requires to store (valid, i.e., not-expired) MX records together with recipient domains. Another example is a list of recipients that wait for a host to be able to receive mail again, i.e., a DB which is keyed on hosts (IP addresses or names?) and the data is a list of recipients for those hosts.

Normally entries in the incoming queue are moved into the deferred queue only after a delivery attempt, i.e., via the active queue. However, the active queue itself is not backed up on persistent storage. Hence an envelope must be either in the incoming queue or in the deferred queue at any given time (unless it has been completely delivered). Moving an entry into the deferred queue must be done safely, i.e., the envelope must be safely in the deferred queue before it is removed from the incoming queue. Question: When do we move the sender data over from the incoming to the deferred queue? Do we do it as soon as one recipient has been tried or only after all have been tried? Since we are supposed to try delivery as soon as we have the mail, we probably should move the sender data after we tried all recipients. "Trying" means here: figure out a DA, check flags/status (whether to try delivery at all, what's the status of the system to which the mail should be delivered), if normal delivery: schedule it, otherwise move to the deferred queue.

The in-memory queues are limited in size. These sizes are specified in the configuration file. It is not yet clear which form these specifications may have: amount of memory, amount of entries, percentage of total memory that can be used by sendmail X or specific processes. The specification should include percentages at which the behavior of the queue manager changes, esp. for the incoming queue. If the incoming queue becomes almost full the acceptance of messages must be throttled. This can be done in several stages: just slow down, reduce the number of concurrent incoming connections (just accept them slower), and in the extreme the SMTP server daemons will reject connections. Similarly the mail acceptance must be slowed down if the active queue is about to overflow. Even though the queue manager will normally favor incoming mail over other (e.g., deferred) mail, it must not be possible to starve those other queues. The size of the active queue does not need to be a direct feedback mechanism to the SMTP daemon, it is sufficient if this is happening indirectly through the incoming queue (which will fill up if items can't be moved fast enough into the active queue). However, this may not be intended, maybe we want to accept messages for some limited time faster than we can send them.

It might be nice for the in-memory queues to vary in size during runtime. In high-load situation those queues may grow up to some maximum, but during lower utilization they should shrink again. Maximum and minimum sizes should be user-configurable. However, in general the OS (VM system) should solve the problem for us.

2.4.1.1 Current Selection of Queues and their Names

Here's the list of queues that are currently used:

One proposal for a set of possible queues is:

- incoming: queue (memory) in which currently incoming envelope data is stored (IQDB), backed up on disk (IBDB); both together are referred to as INCEDB.
- active: queue (memory) from which deliveries are scheduled (AQ or ACTEDB).
- deferred: queue (disk) in which envelope data is stored which refers to recipients that could have not been delivered due to problems or queueing (administrative) decisions (DEFEDB). This queue is also called "main" (or "delayed") queue. It also holds data for DSNs.

There's currently no decision about the queue for corrupted entries.

2.4.2 Queue Formats

The incoming queue must be backed up on stable storage to ensure reliability. The most likely implementation right now is a logfile, in which entries are simply appended since this is the fastest method to store data on disk. This is similar to a log-structured filesystem and we should take a look at the appropriate code. However, our requirements are simpler, we don't need a full filesystem, but only one file type with limited access methods. There must be a cleanup task that removes entries from the backup of the incoming queue when they have been taken care of. For maximum I/O throughput, it might be useful to specify several logfiles on different disks.

The other queues require different formats. The items on hold are only released on request (e.g., for ETRN). Hence they must be organized in a way that allows easy access per domain (the ETRN argument) or other criteria, e.g., a hold message for quarantined entries.

The delayed queue contains items that could not be delivered before due to temporary errors. These are accessed in at least two ways:

1. when a delivery agent is able to instantiate a connection to a destination that was previously unavailable, then it might be appropriate to send other entries in the queue to this location too.
2. items in the queue must be tried periodically. In this case it should be possible to access the entries based on certain criteria, of which possibly the most important is "next time to try".

2.4.2.1 Reference Counting

The queue manager needs to keep a reference count for an envelope to decide when an entry can be removed. This may become non-trivial in case of aliases (expansion). If the LDA does alias expansion then one question is whether it injects a new mail (envelope) with a new body. Otherwise reference counting must take care of this too.

The MAIL (sender) data, which includes the reference counter, is stored in the deferred queue if necessary, i.e., as long as there are any recipients left (reference count greater than zero). Hence we must have a fast way to access the sender data by transaction id. At any time the envelope sender information must be either in the incoming queue or in the deferred queue.

Problem: mailing lists require to create new envelope sender addresses, i.e., the list owner will be the sender for those mails. An e-mail can be addressed to several mailing lists and to "normal" recipients, hence this may require to generate several different mail sender entries. Question: should the reference counter only be in the original sender entry and the newly created entries have references to that? Distributing the reference count is complicated. However, this may mean that a mail sender entry stays around even though all of its (primary) recipients have been taken care of.

It might be necessary to have a garbage collection task: if the system crashes during an update of a reference counter the value might become incorrect. We must assure that the value is never too small, because we could remove data that is still needed. If the stored value is bigger than it should be, the garbage collection task must deal with this (compare fsck for file systems).

2.4.3 Data Flow: Moving Envelopes between Queues

Envelopes received by the SMTP servers are put into the incoming queue and backed up on disk. If an envelope has been completely received, the data is copied into the active queue unless that queue is full[16]. Entries in the active queue are scheduled for delivery. If delivery attempts are done, the results of those attempts are written to the incoming queue (mark it as delivered) or deferred queue as necessary. Entries from the deferred queue are copied into the active queue based on their "next time to try" time stamp.

2.4.3.1 Terminology

It would be nice to define some terms for

- destination host (which hosts are meant? MX records + A records)
- delivery attempt (what exactly is that?)
  - address resolution
  - telling a DA to try delivery to a certain host
  - depending on the result of the previous action either try the next host or record some kind of success or (temporary) error
- delivery failure:
  - address resolution problem
  - to a single host
    - session error
    - transaction error; various stages of a transaction, compare phases of SMTP transaction
  - entire failure (all destinations tried)

This would make it easier to talk about the topics and have a common understanding for readers. *Todo*: Check the RFCs and other literature.

[16] If AQ is full the transaction is currently rejected with a temporary error. This is an area for possible improvements.

2.4.3.1.1 Delivery Attempt A delivery attempt consists of:

1. resolving the recipient address; this can cause a temporary or a permanent failure which must be handled properly, e.g., as if actually sending the mail failed in the same way.
2. trying all (or at least several) destination hosts for a recipient. Which recipient destinations are actually tried depends on the scheduler and the connection caches. Since there might be multiple destination hosts for a recipient, multiple connection attempts might be made. Depending on the failure mode the status of a recipient may need to be updated in the active queue temporarily, and the scheduler determines whether more destination hosts should be tried. Only after all destinations hosts have been tried (or delivery succeeded or permanently failed) the status in other queues will be updated accordingly.

2.4.3.2 Data Flow: Queue oriented View

This section explains how data is added to the various queues, what happens with it, and under which circumstances data is read from a queue if there is no queue into which the data is read, i.e., this is a consumer oriented view.

1. Into IQDB: data comes into IQDB only from the SMTP servers (via QMGR):
   (a) the envelope sender is stored in the incoming queue (IQDB).
   (b) the envelope recipients are stored in the incoming queue (IQDB) and IBDB.
   (c) the content database information is stored in the incoming queue (IQDB) when the transaction is closed; this also causes the transaction data to be written to IBDB.
2. Into IBDB: data comes into IBDB only from the SMTP servers (via QMGR):
   (a) envelope recipients are stored in IBDB.
   (b) when a transaction is closed its data (envelope sender and CDB id) is written to IBDB.
   (c) before mail reception is acknowledged the entire transaction data is committed to IBDB (persistent storage), i.e., the appropriate function (for Unix: `fsync(2)`) is called to make sure the data is actually safely written to persistent storage (usually on disk).
3. Into AQ: data comes into the active queue from two sources:
   (a) data in the incoming queue (IQDB) is copied into the active queue (AQ) when an SMTP server transaction is closed[17,18].
   (b) entries are read from the deferred queue (DEFEDB) based on some criteria determined by the scheduler.
4. From AQ: data in the active queue is taken care of by the scheduler and a cleanup task (for case 4e):
   (a) recipient addresses are sent to the address resolver.

[17] *Question*: copy or move, i.e., should the data be removed from IQDB as soon as it is in AQ? Answer: copy; the data in IQDB is still required for the list of open transactions.

[18] Note: currently the data is copied from IQDB to AQ immediately after the transaction is closed, not when the transaction data is actually committed to IBDB. This is an area where optimizations are possible, e.g., data is not copied into AQ if the destination is down. However, currently data can be transferred to DEFEDB only via AQ, see Section 2.4.3.3.

(b) results from the address resolver are used to update recipients.

(c) delivery transactions consisting of one or more recipients are created based on various criteria and sent to delivery agents.

(d) delivery status received from delivery agents are used to update the data in the various queues (see Section 2.4.3.4 for details).

(e) recipient addresses which are waiting too long for a result from AR or DA must be removed from AQ, they are put into DEFEDB with a temporary error, unless the overall queue return timeout is exceeded.

5. Into DEFEDB:

(a) data is written into DEFEDB whenever a delivery attempt failed.

(b) data is removed from DEFEDB when a delivery attempt succeeded and the data was in DEFEDB before.

(c) In case of a restart IBDB is read and transactions which are not closed are added to the main queue (DEFEDB).

2.4.3.3 Detailled Data Flow: Transaction/Recipient oriented View

This section gives a bit more details about the data flow than the previous section. It does only deal with data that is stored by QMGR in some queue, it does not specify the complete data flow, i.e., what happens in the SMTP server or the delivery agents.

1. the envelope sender (MAIL) is stored in the incoming queue (IQDB).

2. the envelope recipients (RCPT) are stored in the incoming queue (IQDB) and in IBDB.

3. When the final dot is received the content database information is stored in the incoming queue (IQDB), and the transaction data (envelope sender and CDB id) is stored in IBDB. The data is copied into AQ unless AQ is full in which case a temporary error is returned.

4. before the mail reception is acknowledged the entire transaction data is safely committed to the backup of the incoming queue on disk (IBDB).

5. recipient addresses in AQ are sent to the address resolver.

6. results from the address resolver are used to update recipients in AQ. Possible results include (temporary) errors in which case the appropriate actions as explained in Section 2.4.3.4 are taken.

7. delivery transactions consisting of one or more recipients are created based on various criteria and sent to delivery agents.

8. delivery status received from delivery agents is used to update the data in the various queues (see Section 2.4.3.4 for details).

9. recipient addresses which are waiting too long for a result from AR or DA must be removed from AQ, they are put into DEFEDB with a temporary error, unless the overall queue return timeout is exceeded.

10. data from DEFEDB is used to feed the active queue; entries are read from it based on their "next time to try" (or whatever criteria the scheduler wants to apply).

2.4.3.3.1 Incoming Queue: How long to keep data? *Question*: should we keep entries in the incoming queue only during delivery attempts, or should we move the envelope data into the deferred queue while the attempts are going on? If we move the envelopes, we have more space available in the incoming queue and can accept more mail. However, moving envelopes costs of course performance. In the "normal" case we don't need the envelope data in the deferred queue, i.e., if delivery succeeds for all recipients and no SUCCESS DSNs are requested, we don't need the envelope data ever in the deferred queue. Question: do we want a flexible algorithm that moves the envelope data only under certain conditions? Those conditions could include how much space is free in the incoming queue and how long an entry is already in the queue. There should be two different modes (selectable by an option):

- an envelope stays in the INCEDB until the first delivery attempt for each recipient has been made,
- an envelope stays in the INCEDB up to some time limit and some maximum filling level of the RSC has been reached, then it is moved to the DEFEDB (and the data in the active queue is modified accordingly).

We need the envelope data in the deferred queue, if and only if

- a DSN must be sent, i.e., the conditions for sending a DSN are met (theoretically we could try to handle that in the INCEDB too, but that seems to be too complicated in the general case, see Section 2.4.6), or
- a delivery attempt (this includes address resolving) temporarily fails.

If no DSN must be sent and all recipients have been taken care of, the envelope does not need to be moved into the DEFEDB, and it can be removed from the INCEDB afterwards without causing additional data moving.

Note: it should be possible to remove recipient and transaction data from IQDB as soon as it has been transferred to AQ and safely committed to IBDB; at this moment the data is in persistent storage and it is available to the scheduler, hence the data is not really needed anymore in IQDB. There are some implementations issues around this[19], hence it is not done in the current version, this is something that should be optimized in a subsequent version.

2.4.3.4 Updating Data in Queues after a Delivery Attempt

When a delivery attempt (see 4d in Section 2.4.3.2) has been made, the recipient must be taken care of in the appropriate way. Note that a delivery attempt may fail in different stages (see Section 2.4.3.1), and hence updating the status of a recipient can be done from different parts of QMGR. That is, in all cases the recipient address is removed from ACTEDB and

1. for a successful delivery attempt the data in the queue from which the recipient has been placed in ACTEDB must be updated (removed, unless a SUCCESS DSN is requested).
   (a) If it is from INCEDB, remove the entry from INCEDB. This involves updating IBDB (and IQDB if the entry is not removed earlier on).
   (b) If the entry is from DEFEDB, update the counters in the transaction context and remove the recipient entry. When all references to a CDB entry have been removed, then that entry must be removed too.

[19]2004-04-14: Due to way group commits to IBDB are handled and how entries are marked "done" in IBDB.

2. for a temporary delivery failure

   (a) the recipient address is moved to DEFEDB if it was in INCEDB before. This should be done as follows: write an entry to DEFEDB, if the transaction isn't in DEFEDB yet: write transaction and recipient record. Thereafter update INCEDB. It is possible to update INCEDB before DEFEDB if the recovery program is made aware of this, i.e., it has to check DEFEDB whether the data is actually there. However, this is ugly in the case when the recipient has been delivered later on because then this causes double delivery. Hence the proposed order should be maintained.

   (b) otherwise the status of the recipient address in DEFEDB is updated.

3. for a permanent delivery failure

   (a) the recipient address is moved to DEFEDB if it was in INCEDB before (for DSN FAILURE if requested); see item 2a above.

   (b) otherwise the status of the recipient address in DEFEDB is updated (for DSN FAILURE if requested).

   The data is stored in DEFEDB (persistent storage) to avoid retrying a failed delivery, see also Section 2.4.6.

Notice: it is recommended to perform the update for a delivery attempt in one (DB) transaction to minimize the amount of I/O and to maintain consistency. Furthermore, the updates to DEFEDB should be made before updates to INCEDB are made as explained in Section 2.4.1.

Note: this section does not discuss how to deal with a transaction whose recipients are spread out over INCEDB and DEFEDB. For example, consider a transaction with two recipients, all data is in INCEDB. A delivery attempt for one recipient causes a temporary failure, the other recipient is not tried yet. Now the transaction and the failed recipient are written to DEFEDB. However, the recipient counters in the transaction do not properly reflect the number of recipients in DEFEDB but in both queues together. The recovery program must be able to deal with that.

2.4.3.5 Reading Entries from Deferred Queue

According to item 10 in Section 2.4.3.3 entries are read from the deferred queue into the active queue *based on their "next time to try" (or whatever criteria the scheduler wants to apply).* Instead of reading through the entire DB — which is on disk and hence *expensive* disk I/O is involved — each time entries should be added, an in-memory cache (EDBC) is maintained which contains references to entries in DEFEDB sorted based on the "next time to try". Note: it might be interesting to investigate whether an DEFEDB implementation based on Berkeley DB would make this optimization superfluous because Berkeley DB maintains a cache anyway. However, it is not clear which data the cache contains, most likely it is not "next time to try" but only the key (recipient/transaction identifiers).

2.4.3.5.1 Memory Usage Even though each entry in the cache is fairly small (recipient identifier, next time to try, and some management overhead), it might be impossible to hold all references in memory because of the size. Here is a simple size estimation: an entry is about 40 bytes, hence 1 million entries require 40 MB. If a machine actually has 1 million entries in its deferred queue then it has most likely more than 1 GB RAM. Hence it seems fairly unlikely to exceed the available memory with EDBC. Nevertheless, the system must be prepared to deal with such a resource shortage. This can be done by changing into a different mode in which DEFEDB is regularly scanned and entries are inserted to EDBC such that *older* entries will be removed if *newer* entries are inserted and EDBC is full[20].

2.4.3.5.2 Fairness If the MTS is busy it might not be possible to read all entries from DEFEDB when their next time to try is actually reached because AQ might be full. Hence it is necessary to establish some fairness between the two producers for AQ: IQDB (SMTP servers) and DEFEDB. A very simple approach is to reserve a certain amount, e.g., half, for each of the producers. However, that does not seem to be useful:

1. The normal behavior should be that a mail is delivered directly without touching DEFEDB.
2. A static assignment is inflexible.

A slightly better approach is as follows:

1. Reserve a certain amount (e.g., ten per cent) of AQ for entries from DEFEDB.
2. Make sure that no more than some upper threshold (e.g., seventy per cent) of AQ entries are from DEFEDB.

That is, reserve only a minimum space in AQ for both producers and let them freely use the rest of the available space up to their limits. The reserved space for each of them should reflect the expected rate of data generated between the two producers.

This approach will

1. never completely starve DEFEDB (because some amount is reserved for it),
2. allow DEFEDB to use more of AQ if space is available, and
3. never starve IQDB.

2.4.3.6 Cut-Through Delivery

sendmail 8 provides a delivery mode called *interactive* in which a mail is delivered before the server acknowledges the final dot. An enhanced version of this could be implemented in sendmail X, i.e., try immediate delivery but enforce a timeout after which the final dot is acknowledged. A timeout is necessary because otherwise clients run into a timeout themselves and resend the mail which will usually result in double deliveries.

This mode is useful to avoid expensive disk I/O operations; in a simple mode at least the `fsync(2)` call can be avoided, in a more complicated mode the message body could be shared directly between SMTP server and delivery agent to even avoid creation of file on disk (this could be accomplished by using the buffered file mode from sendmail 8 with large buffers, however, this requires some form of memory sharing[21]). Various factors can be used to decide whether to use interactive delivery, e.g., the size of the mails, the number of recipients and their destinations, e.g., local versus remote, or other information that the scheduler has about the recipient hosts, e.g., whether they are currently unavailable etc.

Cut-through delivery requires a more complicated protocol between QMGR and SMTP server. In normal mode the SMTP server calls `fsync(2)` before giving the information about the mail to QMGR and then

[20] *older* and *newer* refer to "next time to try", not when entries are added to EDBC.
[21] System V shared memory, maybe `mmap(2)`, others?

waits for a reply which in turn is used to inform the SMTP client about the status of the mail, i.e., the reply to the final dot. For cut-through delivery the SMTP server does not call `fsync(2)` but informs QMGR about the mail. Then the following cases can happen:

1. QMGR can return one of the following replies:
   (a) accept without `fsync(2)`: the mail has been successfully delivered to all recipients.
   (b) accept with `fsync(2)`: the mail has not been successfully delivered to all recipients.
   (c) reject
2. QMGR does not reply within the timeout: return a temporary error to the client.

For case 1b the SMTP server needs to send another message to QMGR telling it the result of `fsync(2)`. If `fsync(2)` fails, the message must be rejected with a temporary error, however, QMGR may already have delivered the mail to some recipients, hence causing double deliveries.

2.4.4 Scheduling

Items from the delayed queue need to be read into the active queue based on different criteria, e.g., time in queue, time since last attempt, precedence, random walk.

The queue manager must establish a fair selection of items in the incoming queue and items in the delayed queue. This algorithm can be influenced by user settings, which includes simple options (compare QueueSortOrder in sendmail 8), table driven decisions, e.g., no more than `N` connections to a given host, and a priority (see Section 2.4.4.1). A simple way to achieve a fair selection is to establish a ratio (that can be configured) between the queues from which entries are read into the active queue, e.g., incoming: 5, deferred: 1. Question: do we use a fixed ratio between incoming and deferred queue or do we vary that ratio according to certain (yet to determine) conditions? These ratios are only honored if the system is under heavy load, otherwise it will try to get as many entries into the active queue as possible (to keep the delivery agents busy). However, the scheduler will usually not read entries from the deferred queue whose next time to try isn't yet reached, unless there is a specific reason to do so. Such a reason might be that a connection to the destination site became available, an ETRN command has been given, or deliver is forced by an admin via a control command. Question: does the ratio refer to the number of recipients or the number of envelopes?

The QMGR must at least ensure that mail from one envelope to the same destination site is send in one transaction (unless the number of recipients per message is exceeded). Hence there should be a simple way to access the recipients of one envelope, maybe the envelope id is a key for the access to the main queue. See also 2.4.4.4 for further discussion. Additionally MX piggybacking (as in 8.12) should be implemented to minimize the required number of transactions.

Question: how to schedule deliveries, how to manage the active queue? Scheduling: Deliveries are scheduled only from the active queue, entries are added to this queue from the incoming queue and from the deferred queue.

To reduce disk I/O the active queue has two thresholds: the maximum size and a low watermark. Only if too few entries are in the cache entries are read from the deferred queue. Problem: entries from the incoming queue should be moved as fast as possible into the active queue. To avoid starvation of deferred entries a fair selection must be made, but this must be done on a "large" scale to minimize disk I/O. That is, if the ratio is 2-1 (at least one entry from the deferred queue for every two from the incoming queue), then it could be that 100 entries are moved from the incoming queue, and then 50 from the deferred queue. Of course the algorithm must be able to deal with border conditions, e.g., very few incoming entries but large deferred queue, or only a few entries trickling in such that the number of entries in the active queue is always in the range of the low watermark.

Question: where/when do we ask the address resolver for the delivery tuple? That's probably a configuration option. The incoming queue must be able to store addresses in external and in "resolved" form. See also Section 3.13.6 for possible problems when using the resolved form.

Here's a list of scheduling options people (may) want (there are certainly many more):

- initial-destination-concurrency, max-destination-concurrency: how many concurrent deliveries to a site are allowed? This can be further subdivided per site, delivery agent, etc.

- mailinglist-resources: percentage how much scheduling resources can be used by a mailing list expansion (can it use all available delivery slots?).

- site-resources: percentage how much scheduling resources can be used by a single site (can it use all available delivery slots?).

- Retry schedule: by default, the time between retries is doubled after each (temporarily) failed delivery attempt. There are lower and upper limits for the retry time. It must be possible to specify other retry schedules (see below), which may be selected based on several criteria, most likely the destination site. postfix: the next time for a delivery attempt is current time plus time in queue (which is current time minus arrival time). Notice: this will cause an additional delay if the delivery is not attempted at the next retry time but later on (due to load conditions etc).

- Send big mails only at certain times. Maybe this should be: Temporarily defer delivery of large email, looking for a low-usage time. However, there might be administrative restrictions.

Question: how to specify such scheduling options and how to do that in an efficient way? It doesn't make much sense to evaluate a complicated expression each time the QMGR looks for an item in the deferred queue to schedule for delivery. For example, if an entry should only be sent at certain times, then this should be "immediately" recognizable (and the item can be skipped most of the time, similar to entries on hold).

Remark: `qmail-1.03/THOUGHTS` [Ber98] contains this paragraph:

> Mathematical amusement: The optimal retry schedule is essentially, though not exactly, independent of the actual distribution of message delay times. What really matters is how much cost you assign to retries and to particular increases in latency. qmail's current quadratic retry schedule says that an hour-long delay in a day-old message is worth the same as a ten-minute delay in an hour-old message; this doesn't seem so unreasonable.

Remark: Exim [Haz01] seems to offer a quite flexible retry time calculcation:

> For example, it is possible to specify a rule such as 'retry every 15 minutes for 2 hours; then increase the interval between retries by a factor of 1.5 each time until 8 hours have passed; then retry every 8 hours until 4 days have passed; then give up'. The times are measured from when the address first failed, so, for example, if a host has been down for two days, new messages will immediately go on to the 8-hour retry schedule.

Courier-MTA has four variables to specify retries:

retryalpha, *retrybeta*, *retrygamma*, *retrymaxdelta*

These control files specify the schedule with which Courier tries to deliver each message that has a temporary, transient, delivery failure. *retryalpha* and *retrygamma* contain a time interval, specified in the same way as queuetime. *retrybeta* and *retrymaxdelta* contain small integral numbers only.

Courier will first make *retrybeta* delivery attempts, waiting for the time interval specified by *retryalpha* between each attempt. Then, Courier waits for the amount of time specified by *retrygamma*, then Courier will make another *retrybeta* delivery attempts, *retryalpha* amount of time apart. If still undeliverable, Courier waits $retrygamma * 2$ amount of time before another *retrybeta* delivery attempts, with *retryalpha* amount of time apart. The next delay will be $retrygamma * 4$ amount of time long, the next one $retrygamma * 8$, and so on. *retrymaxdelta* sets the upper limit on the exponential backoff. Eventually Courier will keep waiting $retrygamma * (2^{retrymaxdelta})$ amount of time before making *retrybeta* delivery attempts *retryalpha* amount of time apart, until the queuetime interval expires.

The default values are:

- *retryalpha* - five minutes
- *retrybeta* - three times
- *retrygamma* - fifteen minutes
- *retrymaxdelta* - three

This results in Courier delivering each message according to the following schedule, in minutes: 5, 5, 5, 15, 5, 5, 30, 5, 5, 60, 5, 5, then repeating 120, 5, 5, until the message expires.

2.4.4.1 Two Level Scheduling

There are two levels of scheduling:

1. Putting items from the incoming and the deferred queue into the active queue.
2. Sending items from the active queue to the delivery agents.

We could assign each entry a priority that is dynamically computed. For example, the priority could incorporate:

- The time for the next delivery; i.e., the "overdue" time, it should have been tried at time X, but now it is already X+n.
- The number of delivery attempts.
- The time in the queue.
- The number of open connections to the recipient site: 0 should have no influence on the priority, 1 have a positive influence, large values have a negative influence.
- The number of messages delivered to the recipient site since those connections have been opened.
- The message size (negative).
- The total number of recipients (negative).

- The number of recipients to the same site (positive).
- The precedence.
- A random component.
- Deliver-by time specification.

However, it is questionable whether we can devise a formula that generates the right priority. How do we have to weight those parameters (linear functions?), and how to combine them $(+, -, /, *, ...)$? It might be simpler (better) to specify the priority in some logical formula (if-then-else) in combination with arithmetic. Of course we could use just arithmetic (really?) if we use the right operations. However, we want to be able to short-cut the computation, e.g., if one parameter specifies that the entry certainly will not be scheduled now. For example: if time-next-try > now then Not-Now unless connections-open(recipient-site) > 0.

2.4.4.1.1 Low Volume On system with low mail volume the schedulers will not be busy all the time. Hence they should sleep for a certain time (in sendmail 8 that's the `-q` parameter). However, it must be possible to wake them up whenever necessary. For example, when a new mail comes in the first level scheduler should be notified of that event such that is can immediately put that mail into the active queue if that is possible, i.e., there is enough free space. The sleep time might be a configurable option, but it should also be possible to just say: wake up at the next retry time, which is the minimum of the retry times in the deferred queue.

2.4.4.1.2 Next Retry Time The next retry time should not be computed based on the message/recipient, but on the destination site (Exim does that). It doesn't make much sense to contact a site that is down at random intervals because different messages are addressed to it. Since the status of a destination site is stored in the connection cache, we can use that to determine the next retry time. However, we have the usual problem here: a recipient specifies an e-mail address, not the actual host to contact. The latter is determined by the address resolver, and in general, it's not a single host, but a list of hosts. In theory, we could base the retry time on the first host in the list. However, what should we do if another host in the list has a different next retry time, esp. an earlier one? Should we use the minimum of all retry times? We would still have to try the hosts in order (as required by the standard), but since a lower priority host may be reachable, we can deliver the mail to it. Question: under which circumstance can a host in the list have an earlier retry time? This can only happen if the list changes and a new host is added to it (because of DNS changes or routing changes). In that case, we could set the retry time for the new host to the same time as all the other hosts in the list. However, this isn't "fair", it would penalize all mails to that host. So maybe it is best to use the retry time of the first host in the list as the retry time of a message.

Note: There are benefits to some randomness in the scheduling. For example, if some systematic problem knocks down a site every 3 hours, taking 15 minutes to restore itself, then delivery attempts should not accidentally synchronize with the periodic failures. Hence adding some "fuzz" factor might be useful.

2.4.4.1.3 Preemptive Scheduler Notice: it might be useful to have a pre-emptive scheduler. That is, even if the active queue is full, there might be reasons to remove entries from it and replace them with higher priority entries from the incoming queue. For example, the active queue may be filled with a lots of entries from a mailing list and new mail is coming in. If the delivery is slow, then some of those new entries may replace entries in the active queue that aren't actually given to a delivery agent. Theoretically, this could be handled by priorities too.

2.4.4.2 First Level Scheduling (Global)

Whenever there is sufficient free space (number of entries falls below low watermark), then the first level scheduler must put some entries from the incoming and the deferred queue into the active queue.

2.4.4.2.1 Scheduling Large Mailing Lists Problem: we have to avoid using up all delivery agents (all allowed connections) for one big e-mail, e.g., an e-mail to a mailing list with thousands of recipients. Even if we take the number of recipients into account for the priority calculation, we don't want to put all recipients behind other mails with fewer recipients (do we?). This is just another example how complicated it is to properly calculate the priority. Moreover, expansion of an alias to a large list must be done such that it doesn't overflow the incoming queue. That is: where do we put those expanded addresses? We could schedule some entries immediately and put others into the deferred queue (which doesn't have strict size restrictions).

2.4.4.2.2 Immediate Delay Entries from the incoming queue are placed into the active queue in FIFO order in most cases.

Question: do we put an entry from the incoming queue into the active queue even though we know the destination is unavailable or do we move it in such a case directly to the deferred queue? We could add some kind of probability and a time range to the configuration (maybe even per host). Get a random number between 0 and 100 and check it against the specified probability. If it is lower try the connection anyway. Another way (combinable?) is to specify a time range (absolute or as percentage) and check whether the next time to try is within this range.

2.4.4.3 Second Level Scheduling (Micro)

Whenever new entries are added to the active queue, a "micro scheduler" arranges those in an appropriate order. Question: how to do micro scheduling within the active queue?

1. strict FIFO
2. may move entries for open connections up front. In this case we have to make sure that other entries don't "starve". Can global scheduling prevent this problem? It may, but we probably can't rely on that, esp. if that scheduler can be easily tweaked by user configuration.

Question: do we treat the active queue strictly as queue? Probably not because we want to reuse open connections (as much as allowed by the configuration). So if we have an open connection and we move an entry "up front" to reuse the connection, how do we avoid to let other entries "lie around" forever in the queue? We could add a penalty to this connection (priority calculcation), such that after some usages the priority becomes too bad and hence entries can't leapfrog others anymore. The problem is still the same: how to properly calculate the priority without causing instabilities? Courier moves the oldest entry (from the tail) to the head of the queue in such a case to prevent starvation. Question: does this really prevent starvation or is it still possible that some entries may stay in the queue forever?

The second level scheduler must be able to preempt entries in the queue. This is required at least for entries that are supposed to be sent to a destination which turns out to be unavailable after the entry has been placed in the active queue. This can happen if an "earlier" entry has the same destination and that delivery attempt fails. Then all entries for the same destination will be removed from active queue. In such a case, they will be marked as deferred (assuming it was a temporary delivery failure). Notice: this is complicated due to the possibility of multiple destination sites, so all of them have to be unavailable for this to happen. It may also be useful to just remove entries from the active queue based on request by the first level scheduler. Question: how can this be properly coordinated?

2.4.4.4 Minimizing Number of Transactions

As described in Section 2.4.4 the scheduler should at least ensure that mail from one envelope to the same destination site is sent in one transaction (unless the number of recipients per message is exceeded). However, this isn't as trivial to achieve as it seems on first sight. If MX piggybacking is supposed to implemented then all addresses of one envelope must be resolved first before any delivery is scheduled. This may reduce throughput since otherwise delivery attempts can be made as soon as a recipient address is available. If those recipient addresses would be for different destinations then starting delivery as soon as possible is more efficient (assuming the system has not yet reached its capacity). If the recipient addresses are for the same destination then putting them into one transaction will at least reduce the required bandwidth (and depending on the disk I/O system and its buffer implementation maybe also the number of disk I/O operations). Recipient coalescing based on the domain parts is easier to implement since it can be done before addresses are resolved; it still requires walking through the entire recipient list of course (some optimized access structure, e.g., a tree with sublists, could be implemented). Depending on when addresses are resolved and where they are stored, MX piggybacking may be as easily to achieve, i.e., if the resolved addresses are directly available.

2.4.4.5 Cleanup of Active Queue

Entries must not stay in AQ for unlimited time (see Section 2.4.3.2, item 4e) hence some kind of timeout must be enforced. There are two situations in which timeouts can occur:

1. Waiting for AR result.
2. Waiting for DA result.

Theoretically there is a third timeout: A recipient is ready for scheduling but the (micro) scheduler did not yet get to it. This should never happen, it indicates a problem with the scheduler.

2.4.4.6 Outgoing Connection Cache

The queue manager keeps a connection cache that records the number of open connections, the last time of a connection attempt, the status (failure?), etc. For details, see Section 3.4.10.10. Question: if the retry time for a host isn't reached, should an incoming message go directly into the deferred queue instead of being tried? That might be a configuration option. See also 2.4.4.2.2.

2.4.4.7 Multiple Destinations

For SMTP clients, mail might have multiple possible destinations due to the use of MX records. The basic idea is to provide a metric of hosts that are "nearer" to the final delivery host (where usually local delivery occurs). A SMTP client must try those hosts in order of their preference "until a delivery attempt succeeds"[22]. However, this description is at least misleading, because it seems to imply that

[22] To provide reliable mail transmission, the SMTP client MUST be able to try (and retry) each of the relevant addresses in this list in order, until a delivery attempt succeeds. [Kle01], pg 59

if mail delivery fails other destinations hosts should (MUST) still be tried, which is obviously wrong. So the question is: when should a SMTP client (or in this case, the QMGR) stop trying other hosts? One simple reason to stop is of course when delivery succeeded. But what about all the other cases (see Section 3.8.4.1)? qmail stops trying other hosts as soon as a connection succeeded, which is probably not a good idea since the SMTP server may greet with a temporary error or cause temporary errors during a transaction.

2.4.4.8 Data Structures to support Scheduling

The QMGR should maintain the following data structures ("connection caches", "connection databases") to help the scheduler making its decisions:

1. OCC (Open Connection Cache): Currently open (outgoing) connections.
2. DCC (Destination Connection Cache): Previously open (tried) connections.
3. AQRD (AQ Recipients Destination): Destinations of recipients in AQ.

The last structure (3: AQRD) is just one way to access recipients in AQ, in this case via the DA and the next hop ("destination"). It can be used to access recipients that are to be sent to some destination, e.g., to reuse an open connection. All recipients that have the same destination are linked together.

OCC (1) keeps track of the currently open connections and how busy they are as well as the current "load", i.e., the number of open sessions/transactions per destination. This can be used to implement things like slow-start (see 2.4.7) and overall connection limits. Note: these limits should not be implemented per DA, but for the complete MTS. *Question*: should there be only (global) one open connection cache, not one each per DA?

DCC (2) keeps track of previously made/tried connections (not those that are currently open), it can be compared to the hoststatus cache of sendmail 8. This can be used by the scheduler to decide whether trying to connect to hosts at all, e.g., because they are down for some time already.

All three structures are basically accessed via the same key (DA plus next hop); the structures AQRD (3) and OCC (1) keep an accurate state, while DCC (2) might be implemented in a way that some information is lost in order to keep the size reasonable (it is not feasible to keep track of all connections that have ever been made, nor is it reasonable to keep track of all connections for a certain amount of time if that interval is too large, see 3.4.10.10 for a proposal).

Question: is it useful to merge AQRD (3) and OCC (1) together because they basically provide two parts of a bigger picture (and hence merging them avoids having to update and maintain them separately, e.g., memory (de-)allocation and lookups are done twice for each update). However, keeping them separate seems cleaner from a software design standpoint: AQRD is "just" one way to access entry in AQ, while OCC is an overview of the current state of all delivery agents.

There are (at least) two more access methods that are useful for the scheduler:

1. Recipients which are waiting for the AR to determine their destination (DA, next hop). Those are not in any AQ Recipient Destination Queue, unless there is a "pseudo" queue for "unknown" destinations.
2. As explained in Section 2.4.4.5 entries must be removed if they stay too long in AQ.

   It might be useful to organize recipients that are waiting for an AR or DA result into a list which is sorted according to their potential timeout.

2.4.5 Triggering Deliveries

The administrator should have the chance the trigger a delivery attempt or complete queue runs manually. For example, if the admin notices that a site or a network connection is up again after a problem, she should be able to inform the scheduler about this change, see also Section 2.11.1.2.

2.4.6 DSNs

According to RFC 1894 five types of DSNs are possible:

1. "delivered" indicates that the message was successfully delivered to the recipient address specified by the sender, which includes "delivery" to a mailing list exploder; abbreviation: sDSN (success).

2. "expanded" indicates that the message has been successfully delivered to the recipient address as specified by the sender, and forwarded by the Reporting-MTA beyond that destination to multiple additional recipient addresses; abbreviation: eDSN. An action-value of "expanded" differs from "delivered" in that "expanded" is not a terminal state, i.e., further "failed" or "delayed" notifications may be provided.

   Using the terms "mailing list" and "alias" as defined in RFC 2821 [Kle01], section 3.10.1 and 3.10.2: An action-value of "expanded" is only to be used when the message is delivered to a multiple-recipient "alias". An action-value of "expanded" should not be used with a DSN issued on delivery of a message to a "mailing list".

3. "relayed" indicates that the message has been relayed or gatewayed into an environment that does not accept responsibility for generating DSNs upon successful delivery; abbreviation: rDSN.

4. "delayed" indicates that the Reporting MTA has so far been unable to deliver or relay the message, but it will continue to attempt to do so. Additional notification messages may be issued as the message is further delayed or successfully delivered, or if delivery attempts are later abandoned; abbreviation: dDSN. Notice: according to RFC 1891 "delayed" DSN are not required (MAY).

5. "failed" indicates that the message could not be delivered to the recipient; abbreviation: fDSN. The Reporting MTA has abandoned any attempts to deliver the message to this recipient. This DSN type is commonly called "bounce".

The queue manager collects the delivery status informations from the various delivery agents (temporary and permanent failures). Based on the requested delivery status notifications (delay, failure, success), it puts this information together and generates a DSN as appropriate. DSNs are added to the appropriate queue and scheduled for delivery.

Question: how to coalesce DSNs? We don't want to send a DSN for each (failed) recipient back to the sender individually. After each recipient has been tried at least once (see also 3.4.10.6) we can send an initial DSN (if requested) which includes the failed recipients (default setting). Do we need to impose a time limit after which a DSN should be sent even if not all recipients have been tried yet? Assuming that our basic queue manager policy causes all recipients to be tried more or less immediately, we probably don't need to do this. Recipients would not be tried if the policy says so (hold/quarantine), or if the destination host is known to be down and the retry time for each hasn't been reached yet. In these cases those recipients would be considered "tried" for the purpose of a DSN (they are delayed). After the basic warning timeout (either local policy or due to deliver-by) a DSN for the delayed recipients is sent if requested. This still leaves open when to send DSNs for failed recipients during later queue runs. Since the queue manager doesn't schedule deliveries per envelope but per recipient, we need to establish some policy when to send other DSNs. *Todo:* take a look at other MTAs (postfix) how they handle this. Note: RFC 1891, 6.2.8 DSNs describing delivery to multiple recipients: a single DSN may describe attempts to deliver a message to multiple recipients of that message. Hence the RFC allows to send several DSNs, it doesn't require coalescing.

Notice: it is especially annoying to get several DSNs for the same message if the full message is returned each time. However, it would probably violate the RFCs to return the entire mail only once (which could be fairly easily accomplished). BTW: the RET parameter only applies to "failed" DSNs, for others only headers should be returned (RFC 1891, 5.3).

Additional problem: different timeouts for warnings. It is most likely possible to assign different timeouts for DELAY DSNs to different recipients within a single mail. In that case the algorithm to coalesce DELAY DSNs will be even more complicated, i.e., it can't be a simpler counter whether all recipients have been tried already.

Question: where do we store the data for the DSN? Do we store it in the queue and generate the DSN body "on the fly" or do we create a body in the CDB? Current vote is for the former.

2.4.6.1 DSN Recipient Types

A MTA must be able to distinguish between different types of recipient addresses:

1. regular local recipients and mailing lists: those addresses can cause delivered, delayed, and failed DSNs.
2. aliases: those addresses can cause expanded, delayed, and failed DSNs.
3. non local recipients: those addresses can cause relayed, delayed, and failed DSNs.

Note: RFC 1891, 6.2.7.4 explains confidential forwarding addresses which should be somehow implemented in sendmail X.

It doesn't seem to be easy to maintain this data. First of all, the types are only known after address expansion. Even then, they may not be complete because a LDA may perform further alias expansion. Question: must the sum of these counters be the same as the number of original recipients? That is, "all" we have to do is to classify the original recipients into those three cases and then keep track of them? Answer: no. DSNs can be requested individually for each recipient. Hence the question should be: must the sum of these counters be less than or equal the number of original recipients?

The main problem is how to deal with address expansions, i.e., addresses that resolve (via aliases) to others. RFC 1891 lists the following cases:

- mailing list: considered as final delivery, hence no problem.
- single-recipient alias: should propagate DSN parameters.
- multiple-recipient alias: can be treated in different ways:
  1. treat as relaying to system that doesn't support DSN.
  2. propagate DSN parameters to exactly one address. This is for example useful for vacation aliases, i.e.,

     ```
     user
     "|/usr/bin/vacation user"
     ```

3. propagate ENVID, RET, ORCPT, NOTIFY (without SUCCESS) parameters to all addresses, if SUCCESS was requested: “expanded” DSN.

2.4.6.2 DSN: Return Body or Headers

If DSNs are implemented properly the sender can determine herself whether she wants the full body or just the headers of her e-mail returned in a DSN. sendmail 8 has a configuration option to not return the body of an e-mail in a bounce (to save bandwidth etc). In addition to that, it might be useful to have a configuration option to return the body only in the first bounce but not in subsequent DSNs (see Section 2.4.6 about the problem to send only one DSN). So at least two options are necessary:

1. return full body [default]
2. return full body on first bounce, headers on subsequent bounces
3. return only headers on all bounces

These options need to be combined with the DSN requests such that the “minimum” is returned, e.g., if option 2 is selected but the sender requests only headers, then only the headers are sent.

2.4.6.3 DSN: Delayed

Handling a ”delayed” DSN (dDSN) is a bit more complicated than handling ”delivered” cases because the original mail stays in the queue and will be tried again (usually).

There are the following states for a recipient:

1. did not request dDSN.
2. did request dDSN.
   (a) no dDSN generated (yet).
   (b) dDSN generated.
      i. dDSN not yet delivered.
      ii. dDSN delivered.

2.4.7 Load Control

There are two aspects of load control:

1. make sure that the own server doesn’t become overloaded, i.e., deal with high (local) load appropriately.
2. do not overload a remote site when sending mail to it.

Some of the measures can be applied to both cases (local/remote load, incoming/outgoing connections).

The queue manager must control local resource usage, by default it should favor mail delivery over incoming mail. To achieve this, the queue manager needs to keep the state of the entire system or at least it must be able to gather the relevant data from the involved sendmail X programs and the OS. This data must be sufficient to make good decisions how to deal with an overload situation. Local resources are:

- CPU,
- (virtual) memory,
- disk I/O, and disk space,
- network I/O.

Therefore the system state (resource usage) should include:

- CPU: load averages; the calculation of load average is done differently on different OS. On OpenBSD: getloadavg() returns the number of processes in the system run queue averaged over various periods of time. On some Linux versions this number also includes the processes that are waiting for I/O (which means they are blocked). It is certainly interesting to measure the I/O activity (iostat(8)?) because that is also an important indication how busy the system is.
- memory usage (including paging/swapping)
- I/O activity: are the I/O channels saturated? Question: how to measure this? iostat(8): transactions per second, KB per transaction, MB per second. netstat(1): statistics.
- disk space: even though this is more resource control than load control, smX must deal with disk space shortage.
- number of sendmail X processes and their contributions to the load.

This information should be made available via an API such that it can be configured easily according to local requirements. The API should supply more than just a single number ("load"), but values for the different resources that are used as listed above. Question: is that list exhaustive?

The queue manager must be able to limit number of messages/resources devoted to a single site. This applies to incoming connections as well as to outgoing connections. It must also be possible to allow all the time connections from certain hosts/domains, e.g., localhost for submissions. This can be a fixed number or a percentage of the total number of connections or the maximum of both.

The queue manager must assure that the delivery agents do not overload a single site. It should have an adaptive algorithm to use "optimal" number of connections to a single site, these must be within specified limits (lower/upper bound) for site/overall connections. Question: how can the QMGR determine the "optimal" number of connections? By measuring the throughput or latency? Will the overhead for measurements kill the potential gain? Proposal: Check whether the aggregate bandwidth increases with a new connection, or if it stays flat. If connections are refused: back off.

The queue manager may use a "slow start" algorithm (TCP/IP, postfix) which gradually increases the number of simultaneous connections to the same site as long as delivery succeeds, and gradually decreases the number of connections if delivery fails.

Idea (probably not particularly good): use "rate control": don't just check how full the INCEDB is, but also the rate of incoming and "leaving" mails. Problem: how to count those? Possible solution: keep the numbers over certain intervals (5s), count envelopes (not recipients, deal with envelope splitting). If the incoming rate is higher than the departure rate and a certain percentage (threshold) is reached:

slow down mail reception. If the leaving rate is higher than the incoming rate, the threshold (usage of INCEDB) could be increased. However, since more data is removed than added, the higher threshold shouldn't be reached at all. This can only be useful if we have different thresholds, e.g., slow down a bit, slow down more, stop, and we want to dynamically change them based on the rates of incoming and outgoing messages.

All parts of sendmail X, esp. the queue manager, must be able to deal with local resource exhaustion, see also Section 2.15.7.

The queue manager must implement proper policies to ensure that sendmail X is resistant against denial of service attacks. Even though this can't be completely achieved (at least not against distributed denial of service attacks), there are some measure that can be taken. One of those is to impose restrictions on the number of connections a site can make. This applies not only to the currently open connections, but also to those over certains time intervals. For this purpose appropriate connection data must be cached, see Section 3.13.8.

2.4.8 Misc

Todo: structure the following items.

- Question: do we want to allow "hold" per recipient? This seems to be useful and should be fairly easy to achieve.
- limits for number of recipients per message, per transaction (in/out)?
- limits for transactions per session (in/out)?
- transfer message to same host over just one connection (multiple transactions) or multiple? (configurable).
- VERP[23]: maybe optional? The queue manager should be designed to allow for this.

2.4.9 Security Considerations

The queue manager does not need any special privileges. It will run as an unprivileged user.

The communication channels between the various modules (esp. between the QMGR and other modules) must be protected. Even if they are compromised, the worst that is allowed to happen is a local denial of service attack and the loss of e-mail. Getting write access to the communication channel must not result in enhanced privileges. It might sound bad enough that compromising the communicaton channels may cause the loss of e-mail, but consider that an attacker with write access to the mail queue directory may accomplish the same by just removing queued mail. There is one possible way to protect the communication even if an attacker can get write access to them: by using cryptography, i.e., an authenticated and encrypted communication channel (e.g., TLS). However, this is most likely not worth the overhead. It could be considered if the communication is done over otherwise unsecured channels, e.g., a network.

[23]Variable Envelope Return Paths, [Ber97]

2.5 SMTP Server Daemon

There are several alternatives how to implement an SMTP server daemon. However, before we take a look at those, some initial remarks. We have to distinguish between the process(es) that listen(s) for incoming connections (on port 25 by default, in the following we will only write "port 25" instead of "all specified ports") and those processes that actually deal with an SMTP session. We call the former SMTP listener and the latter SMTP server, while SMTP server daemon is used for both of them. It might be that listeners and servers are different processes (passing open file descriptors from the listener to the server) or the same.

2.5.1 Internet Server Application

Interesting references about the architecture of internet servers are: [Keg01] for WWW server models and evaluations, [SV96c], [SV96a], and [SV96b] for some comparisons between C, C++, and CORBA for various server models, and [SGI01] for one particular thread model esp. designed for internet server/client applications. Papers about support of multi-threading by the kernel and in libraries are [ABLL92], [NGP02], [Dre02], and [Sol02].

An internet server application (ISA) reads data from a network, performs some actions based on it, and sends answer(s) back. The interesting case is when the ISA has to serve many connections concurrently. Each connection requires a certain amount of state. This state consists at least of:

- global data that can be shared among all server instances, most of it is read-only, e.g., configuration data, some if it is read/write, e.g., number of open connections and other performance related data. Access to the latter must be synchronized.
- local data relevant for one connection only; this includes data about the connection itself, e.g., file descriptors, I/O buffers, application state (waiting for command, ...), but also data about the thread of execution, e.g., registers, stack, PC, which usually is under control of the OS (or a threading library). It would be nice to minimize the amount of the latter, which can be done by having as few execution threads as possible.

There is a certain amount of hardware concurrency that must be efficiently used: processors (CPU, I/O), asynchronously operating devices, e.g., SCSI (send a command, do something else, get result) and network I/O. There should be one process per CPU assuming the process is runnable all the time (or it invokes a system call that executes without giving up the CPU for the duration of the call); if the process can block then more processes are required to keep the CPU busy. We need to have always one thread of execution that is runnable. Unix provides preemptive, timesliced multitasking, which might not the best scheduling mechanism for ISA purposes. Assuming that context switches are (more or less) expensive, we want to minimize them. This can be achived by "cooperative" multitasking, i.e., context switches occur only when one thread of execution (may) block. Notice: this requires that no thread executes too long such that other threads may starve. This will be a problem if a thread executes a long (compute-intensive) code sequence, e.g., generation of an RSA key. Question: how can we avoid this problem? Maybe use preemptive multitasking, but make the timeslice long enough? As long as each thread only performs a small amount of work, it is better to let it executes its entire work up to a blocking function to minimize the context switch overhead. Question: can we influence the scheduling algorithm for threads? POSIX threads allow for different scheduling algorithms, but most OS implement only one (timesliced, priority based scheduling).

An ISA should answer I/O requests as fast as possible since that allows clients (which are waiting for an answer) to proceed. Hence threads that are able to answer a request should have priority. However, a thread that performs a long computation must proceed too, otherwise its client may time out. So we have a "classic" scheduling problem. Question: do we want to get into these low-level problems or leave it to the OS?

2.5.2 SMTP Server Design Alternatives

The alternatives to implement SMTP server daemons are at least:

1. One process per connection (old sendmail model). This is too slow and too resource intensive (unless the SMTP server is very small). Context switching (between processes) is fairly expensive. Each process occupies a kernel structure.

2. One process (at most per CPU) for all connections (event driven), no threads. This only scales (for a multi-processor system) by starting one process per CPU. However, purely event driven programming is hard (see Section 3.16.4) and long computations (without preemption) will delay all other connections within a process. Hence this model is not acceptable in general.

3. Process with pre-forked children: one process listens on all necessary ports (`select()`), hands over connection to other process (pool of processes available). Similar problems as in 1, it only avoids the (per connection) fork()ing overhead.

4. Threads:

   (a) One thread per connection. This probably doesn't scale very well due to restrictions in current OSs. For example, there might be locking problems, e.g., access to file descriptors (open(), close()). Even though some OS provide a many-to-one mapping between user threads and execution engines, that mapping may detoriate to almost 1-1 because the OS creates a new execution engine if all of them are blocked but there are runnable user threads [SGI01].

   (b) Set of worker threads. This allows to balance the number of worker threads versus the number of tasks to perform and the capabilities of the OS. Problems:

      - Asynchronous I/O. On some OSs the appropriate functions (`select(), poll()`) don't scale very well with the number of file descriptors (events) to watch.
      - This can be fairly complicated to program if it is done purely event driven (see Section 3.16.4); a potential workaround for this problem is explained in Section 3.20.4.1.
      - Many synchronization points have to be build into the program. This can be time-consuming (runtime and programming time), and it is error prone (deadlocks). On multi-processor systems inter-processor synchronization can be fairly expensive. It would be best (but not possible with POSIX threads?) to lock threads down to one processor and deal with multiple processors by having multiple processes (one per processor).

   Disadvantage: threads require very careful programming, the program must never crash, even if running out of memory or "fatal" errors in one of the threads (connections). Only that connection must be aborted and the problem must be logged.

5. Combination of processes and threads (Apache 2): process with pre-forked children which are multi-threaded (worker model); one process listens on all necessary ports for incoming connections and it hands over a new connection to another process (pool of processes available).

   Advantages: "crash resistent" (if one thread goes down, it can take down only one process). The probably most important part of this solution is: it doesn't bind us to any particular model which may show deficiencies on a particular OS, configuration, or even in the long run of further OS development. We can easily tune the usage of processes and threads based on the OS requirements/restrictions. It is the most flexible (and most complicated) model, with which we can get around limitations in different OSs.

Disadvantages: pretty complicated. Selecting this model shouldn't be necessary for crash resistence since we don't have plugins that could easily kill smtpd, but we use external libraries (Cyrus SASL, OpenSSL). It requires extra efforts to share data since we have multiple processes.

Todo: we have to figure out which of those models works best. A comparison [RB01] between 4a, 4b, and one process per connection clearly points to 4b. However, the tests listed in that article are not really representable for SMTP because no messages have been sent. Moreover, it misses model 5. Even though there might be some data available about the performance of different models, most of those probably apply to HTTP servers (Apache) or caches (Squid). These are not really representative for SMTP because HTTP/1.0 is only one request/one response exchange where the response is often pretty short. SMTP uses several exchanges (some of which can be pipelined) and often transports larger data. HTTP/1.1 can be used to keep a connection open (multiple requests and answers) and might be better comparable in its requirements to SMTP. Question: is there performance data available for this? How about FTP (larger data transports, but often only one request)?

Notice: slow connections must be taken into account too. Those connections have very little (I/O, computational) requirements per time slice, but they take up as much context data as fast connections. It should be tried to minimize any additional data, e.g., process contexts, for these connections. If we for example use one thread per connection, then slow connections will take up an entire thread context, but rarely use it. A worker thread model reduces this overhead.

We need some basic prototypes to do comparisons, esp. on different OS to find out whether threading really achieves high performance (probably on Solaris, less likely on *BSD).

Question: if we choose 5 (current favorite), how do we handle incoming connections? Do we have a single process listen on port 25 and handing over the socket to another process (3.16.15)? This may create a bottleneck. Alternative: similar to postfix have multiple processes do a `select()` on the port. One of them will get the connection. Possible problem(s):

- Thundering herd: all processes are woken up by the kernel, but only one gets the connection, the rest an error. We can avoid this problem by passing around a token between the different SMTP listener processes such that only one of them actually performs an accept() on the socket(s). The "best" solution to this problem probably depends on the number of processes and the OS.

- How to dynamically create and destroy processes in this model? Remember: they must share access to the port. This would be done by the MCP.

Question: how much data sharing do we need in SMTPS? We need to know the number of open connections (overall, per host) and the current resource usage. It might be sufficient to have this only in the listener process (if we decide to go for only one) or it can be in the queue manager (which implements general policy and has data about past connections too). Another shared resource is the CDB which may require shared data (locking). If the transaction and session ids are not generated by the queue manager, then these require some form of syncronization too. In the simplest form, the process id might be used to generate unique ids (the MCP may be able to provide ids instead if process ids will not be useful for this purpose because they may be reused too fast).

2.5.3 Control Flow

A SMTP session may consist of several SMTP transactions. The SMTP server uses data structures that closely follow this model, i.e., a session context and a transaction context. A session context contains (a pointer to) a transaction context, which in turn points back to the session context. The data is stored by the queue manager. The transaction context "inherits" its environment from the session context. The session context may be a child of a server daemon context that provides general configuration information. The session context contains for example information about the sending host (the client) and possibly active security and authentication layers.

Todo: describe a complete transaction here including the interaction with other components, esp. queue manager.

The basic control flow of an incoming SMTP connection has already been described in Section 2.1.5.

1. Session: The SMTP server receives an incoming connection attempt and contacts the queue manager (see Section 2.4) with that information (client IP address; address to host name conversion is done by the queue manager or address resolver, for this an asynchronous DNS interface is needed). An ident lookup is performed if requested (configuration option, maybe even per connection). The queue manager and active milters decide whether to accept or reject the connection. The queue manager returns a session id, which will be used at least for logging. In the latter case the status of the server is changed appropriately and most commands are rejected. Further policy decisions can be made, i.e., which features to offer to the client: allow ETRN, AUTH (mechanisms?), STARTTLS (certs?), EXPN, VRFY, etc.

2. HELO/EHLO: send the list of features as returned from the queue manager after the connection has been made. That list of features also includes whether the parameter needs to be checked. If requested, the parameter is sent to the queue manager.

3. If SMTP commands are used that change the status of a session (e.g., STARTTLS, AUTH), those are executed and their effects are stored in the session context. The data is also send to the queue manager and maybe a new list of features is returned. In some cases the SMTP session is started again, e.g., when encryption is turned on. That may change the available features.

4. Transaction: For each transaction a new envelope is created and the commands are communicated to the queue manager and the address resolver for validation and information. Other processes (esp. milters) might be involved too and the commands are either accepted or rejected based on the feedback from all involved processes.

   (a) MAIL: starts a new transaction. The MAIL command is syntactically analyzed, esp. the address and the extensions, and rejected if the syntax check fails. Then the address is given to the anti-UBE checks and the active milters, which may cause a rejection. If some of the extensions are not available or the requested action cannot be taken, the command is rejected too.

   The whole command is passed to the QMGR which stores the relevant data in the incoming queue. Questions: whole or only relevant parts? Are there irrelevant parts? Do we send the original text or a decoded version? A decoded version seems better to avoid double work.

   (b) RCPT: add another recipient to the list. The RCPT command is syntactically analyzed, esp. the address and the extensions, and rejected if the syntax check fails. If some of the extensions are not available or the requested action cannot be taken, the command is rejected too. The address is given to the anti-UBE checks (esp. anti-relay) and the active milters, which may cause a rejection. Moreover, if the recipient is supposed to be local, then it is checked via the appropriate databases, e.g., mailbox and aliases database. The recipient command is sent to the queue manager which adds it to the incoming queue for the current transaction. The queue manager may ask at this point the address resolver to turn the address into an internal form and to expand aliases.

(c) DATA: start collecting data. This is the latest point at which the CDB must be contacted to get an identifier and to open a datafile for writing. The server only adds a `Received:` header and it counts those headers ("trace" headers) for a basic form of loop prevention. It doesn't do any header munging etc. – unless specifically requested – nor does it mess with the body.

Notice: the server must check whether the first line it reads is a header line. If it isn't, it must put a blank line after its `Received:` header as a separator between header and body. If the first line starts with a white space character (LWSP), then a blank must be inserted too. This should be covered by the "is a header" check because a header can't start with LWSP (it would be folded into the previous line).

(d) final dot: When an e-mail is received, the queue manager and the SMTP server must either write the necessary information to stable storage or a delivery agent must take over and deliver the e-mail immediately. The final dot is only acknowledged after either of these actions successfully completed.

(e) RSET (EHLO/HELO): aborts a transaction, all transaction related data must be cleared and the queue manager must be informed about this such that it can remove the transaction from the EDB.

5. QUIT: ends a session.

Other commands (like NOOP, HELP, etc) can be treated fairly simple and are not (yet) described here.

Question: who removes entries from the CDB? Why should it be the SMTP server? The idea from the original paper was to avoid lockings overhead since the SMTP server is the only one which has write access to the CDB. Note: if we use multiple SMTP server processes then we may run into locking issues nevertheless. The QMGR controls the envelope databases which contain the reference counters for messages. Hence it is the logical place to issue the removal command. However, it's still not completely clear which part of sendmail X actually performs the removal.

Misc:

Questions: which storage format should be used? Most likely: network format (CR LF, dot-stuffed). What about BDAT handling?

2.5.4 Anti-Spam Checks

The SMTP server must provide similar anti-spam checks as sendmail 8 does. However, it must be more flexible. Currently it is very complicated to change the order in which things are tested. This causes problems in various situations. For example, before 8.12 it could have been that relaying has been denied due to temporary failures even though the mail could have gone through. This was due to the fixed order in which the checks where run and the checks were stopped as soon as an error occurred even if it was just a temporary error. This has been fixed in 8.12 but it was slightly complicated to do so.

The anti-spam checks belong in the SMTP server. It has all the necessary data, i.e., client connection and authentication (AUTH and STARTTLS) data, sender and recipient addresses. If the anti-spam checks are done by an outside module, all these data need to be sent to it. However, anti-spam checks most likely have to perform map requests, and such calls may block. It might be interesting to "parallelize" those requests, esp. for DNS based maps, i.e., start several of those requests and collect the data later on.

This of course makes programming more complicated, it might be considered as an enhancement later on. We need to define a clean API and then it may be available as library which can be linked into the SMTPS or the AR or another module.

2.5.5 Valid User Checks

The SMTP server must offer a way to check for valid (local) users (see Section 2.6.6). Otherwise mail to local addresses will be rejected only during local delivery and hence a bounce must be generated which causes problems due to forged sender addresses, i.e., they result in double bounces and may clog up the MTS.

2.5.6 Address Rewriting

See also Section 2.6.4.

2.5.6.1 Envelope Address Rewriting

There must be an option to rewrite envelope addresses. This should be separately configurable for sender and recipient addresses.

2.5.6.2 Header Address Rewriting

If sendmail acts as a gateway, it may rewrite addresses in the headers. This can be done by contacting an address rewrite engine. Question: should this be just another mode in which the address resolver can operate?

Question: what is the best way to specify when address rewriting should be used? It might be useful to do this based on the connection information, i.e., when sendmail acts as a gateway between the internal network and the internet.

It would be nice to implement the address rewriting as just another file type. In this case the SMTP servers could just open a file and output the data (message header) to that file. This file type is a layer on top of the CDB file type. The file operations are in this case stateful (similar to those for TLS). As soon as the body is reached, no more interference occurs. Using this approach makes the SMTP server simpler since it doesn't have to deal with the mail content itself.

It must be specifyable to which headers address rewriting is applied. There are be two different classes of header addresses: sender and recipient. These should relate to two different configuration options.

2.5.7 Security Considerations

The SMTP server must bind to port 25, which can be done by the supervisor before the server is actually started provided the file descriptor can be transferred to the SMTP server. sendmail 8 closes the socket if the system becomes overloaded which requires it to be reopened later on, which in turn requires root privileges again.

The SMTP server may need access to an authentication database which contains secret information (e.g., passwords). In most systems access to this information is restricted to the root user. To minimize the exposure of the root account, access to this data should be done via daemons which are contacted via protected communication means, e.g., local socket, message queues.

In some cases it might be sufficient to make secret information only available to the user id under which the SMTP server is running, e.g., the secret key for a TLS certificate. This is esp. true if the information is only needed by the SMTP server and not shared with other programs. An example for the latter might be "sasldb" for AUTH as used by Cyrus SASL which may be shared with an IMAP server.

2.6 Address Resolver

The address resolver (AR) has at least two tasks:

1. determine the delivery information for a recipient address; in sendmail 8/postfix: the mailer triple consisting of delivery agent, next hop, and recipient address in a form suitable for delivery.
2. rewrite an address into an appropriate (canonical) form.

Hence the AR is not just for address resolving but also address rewriting. Other tasks might include anti-spam checks. Question: should the two main tasks (rewriting and resolving) be strictly separated?

Question: what kind of interfaces should the address resolver provide? Does it take addresses in external (unparsed) form and offer various modes, e.g., conversion into internal (tokenized) form, syntax check, anti-spam checks, return of a tuple containing delivery agent, host, address and optionally localpart and extension?

Question: who communicates with the AR? The amount of data the AR has to return might be rather large, at least if it is used to expand aliases (see Section 2.6.7). Using IPC for that could cause a significant slowdown compared to intra-process communication. So maybe the AR should be a library that is linked into the queue manager? Possible problems: other programs need easy access to the AR; security problems since AR and QMGR run with the same privileges in that case? Moreover, the AR certainly performs blocking calls which probably should not be in the QMGR. See also Section 3.6.3.4.

2.6.1 Address Resolver Operation

Usually the address resolver determines the next hop (mailer triple in sendmail 8) solely based on the envelope and connection information and the configuration. However, it might be useful to take also headers or even the mail body into account. Question: should this be a standard functionality of the AR or should this be only achievable via milters (see Section 2.10) or should this not be available at all? Decision: make this functionality only available via milters, maybe not even at all. It might be sufficient to "quarantine" mails (even individual recipients), or reject them as explained in Section 2.10.

2.6.2 Generic Comment about Map Lookups

sendmail 8 uses a two stage approach for most address lookups:

1. Check whether the domain part is in some class (see also Section 4.3.3).
2. If that is the case, then lookup the address (or parts thereof) in some map.

This approach has the advantages that it can avoid map lookups – which may be expensive (depending on the kind of maps) and in most case several variations are checked – if the entry is not in a class. It has the disadvantages that the class data and the map data must be kept in sync, e.g., it is not sufficient to simply add some entries to a map, the domain part(s) must be added to the corresponding class first[24].

2.6.3 Mail Routing

sendmail 8 provides several facilities for mail routing (besides ruleset hacking):

- virtusertable (ruleset 0): changes envelope address. The changed address is used for delivery decisions.
- mailertable (ruleset 0): doesn't change address, just for re-routing.
- SMART-HOST (ruleset 0), MAIL-HUB (ruleset 5), etc: apply in certain situations, may change domain part.

2.6.4 Address Rewriting

- masquerading (applies after routing): changes domain part.
- canonical address mapping (applies after routing): applies to entire address.
- virtusertable (applies before routing): can change entire address, also affects mail routing.

2.6.5 Proposal for Routing and Rewriting

As can be seen from the previous sections, there are operations that solely affect mail routing and there are operations that solely affect address rewriting. However, some operations affect both, because address rewriting is done before mail routing. Hence the order of operations is important. If address rewriting is performed before mail routing, then the latter is affected. If address rewriting is done after mail routing, then it applies only to the address part of the resolved address (maybe it shouldn't be called resolved address since it is more than an address).

Proposal: provide several operations (rewriting, routing) and let the user specify which operations to apply to which type of addresses and the order in which this happens.

Operations can be:

- Rewrite addresses to standard form, more or less the same as ruleset 3 (or Canonify).
- Canonical address mapping, applies to full address; called (misleadingly) genericstable in sendmail 8.
- Hostname masquerading: applies only to domain part.
- Virtual address mapping: applies to full address, can be applied recursively. Maybe it is useful to specify an upper limit for the number of recursions.

[24] which in the case of sendmail 8 even requires a restart of the daemon.

- Mailer selection. For most general operation this should apply to the full address. Special mailers are: error, maybe others like hold, quarantine? The functionality of the latter could also be achieved by special error codes in the range from 6xy to 9xy.

It might be useful for routing operations to not modify the actual address, i.e., if user@domain is specified, it can be redirected to some other host with the original address or with some new address, e.g., user@host.domain.

Some operations – like masquerading – only modify the address without affecting routing.

So for a clear separation it might be useful to provide two strictly separated set of operations for routing and rewriting. However, in many cases both effects (routing and rewriting) are required together.

Address types are: envelope and header addresses, recipient and sender addresses (others), so there are (at least) four combination.

envelope-sender-handling { canonify }

envelope-recipient-handling { canonify, virtual, mailertable }

Question: is this sufficient?

2.6.6 Valid Recipients

It must be possible to specify valid recipients via some mechanism. In most cases this applies to local delivery, but there is also a requirement to apply recipient checks to other domains, e.g., those for which the system allows relaying.

2.6.6.1 Valid Local Recipients

Note that local recipients can often only be found in maps that do not specify a domain part, hence the local domains are separately specified. *Question*: is it sufficient if (unqualified) local recipients are valid for every local domain or is it necessary to have for each local domain a map which specifies the valid recipients? For example, for domain A check map M(A), for domain B check map M(B), etc. Moreover, the domain class would specify whether the corresponding map contains qualified or unqualified addresses. Other attributes might be: preserve case for local part, allow +detail handling, etc.

Configuration example:

```
local-addresses {
  domains = { list of domains };
  map { type=hash, name=aliases, flags={rfc2821}}
  map { type=passwd, name=/etc/passwd, flags={local-parts}}
}
```

2.6.6.2 Valid Remote Recipients

Valid remote recipients can be specified via entries in an access map to allow relaying to specific addresses, e.g.,

```
To:user@remote.domain   RELAY
```

If not all valid recipient are known for a domain for which the MTA acts as backup MX server, then an entry of the form:

```
To:@remote.domain   error:451 Please try main MX
```

should be used.

2.6.7 Aliases

There are different types of aliases: those which expand to addresses and those which expand to files or programs. Only the former can be handled by the address resolver, the latter must be handled by the local delivery agent for security reasons. Note: Courier-MTA also allows only aliases that expand to e-mail addresses, postfix handles aliases in the LDA. If alias expansion is handled by the LDA then an extra round trip is added to mail delivery. Hence it might be useful to have two different types of alias files according to the categorization above.

Problem: if the SMTP server is supposed to reject unknown local addresses during the RCPT stage, then we need a map that tells us which local addresses are valid. There are two different kinds: real users and aliases. The former can be looked up via some mailbox database (generalization of getpwnam()), the latter in the aliases database. However, if we have two different kinds of alias files then we don't have all necessary information unless the address resolver has access to both files. This might be the best solution: the address resolver just returns whether the address is valid. The expansion of non-address aliases happens later on.

The address resolver expands address aliases when requested by the queue manager. It provides also an *owner* address for mailing lists if available. This must be used by the queue manager when scheduling deliveries for those expanded addresses to change the envelope sender address.

The queue manager changes the envelope sender for mailing list expansions during delivery. RFC 2821 makes a distinction between *alias* (3.10.1) and *list* (3.10.2). Only in the latter case the enveloper sender is replaced by the owner of the mailing list. Whether an address is just an alias or a list is a local decision. sendmail 8 uses owner-address to recognize lists.

Question: what to do about delivery to files or programs? For security reasons, these should never end up in the queue (otherwise someone could manipulate a queue file and cause problems; sendmail would have to trust the security of the queue file, which is a bad idea). In postfix aliases expansion is done by the local delivery agent to avoid this security problem. It introduces other problems because no checkpointing will be done for those deliveries (remember: these destinations – they are *not* addresses – never show up in queue files).

Notice: alias expansion can result in huge lists (large number of recipients). If we want to suppress duplicates, we need to expand the whole list in memory (as sendmail 8 does now). This may cause problem (memory usage). Since we can't act the same as older sendmail versions do (crash if running out of memory), we need to restrict the memory usage and we need to use a mechanism that allows us to expand the alias piecewise. One such algorithm is to open a DB (e.g., Berkeley DB; proposed by Murray) on disk and add the addresses to it. This will also detect duplicates if the addresses are used as keys. To avoid double delivery, expansion should be done in the local delivery agent and it must mark mails with a Delivered-To: header as postfix [Ven98] and qmail do. Should attempted double delivery (delivery to a recipient that is already listed as Delivered-To:) in this case cause a DSN? Question: is it ok to list all those Delivered-To: headers in an email? Does this cause an information leak? Question: is ok to use Delivered-To: at all? Is this sanctioned by some RFC? Question: do we only do one level expansion per alias lookup? This minimizes the problem about "exploding" lists, but it may have a significant performance impact (n deliveries for n-level expansion).

Question: should there be special aliases, e.g., ERROR, DEFER, similar to access map, that cause (temporary) delivery errors, or can those be handled by the access map?

2.6.7.1 Forward

Question: Who does `.forward` expansion?

1. Address resolver: it does not (should not) run as root, hence the `.forward` must be group or world readable. If group readable: users must be belong to that group to do a `chgrp`.

2. Delivery agent: a local delivery agent usually needs to run with the privileges of the recipient (started by root, `setreuid(rcpt-id)`). Hence it would be the perfect candidate to access `.forward` files. However, this requires a reinjection of the mail into the system. Can we do this via a simple interface to the queue manager, e.g., some redirect feature? It might be similar (but simpler) to the address resolver interface, i.e., it doesn't return structured information. SMTP allows for an error code to indicate that an address has changed: `551 User not local; please try <new@address>`. This could be used for LMTP too and thanks to continuation lines a whole list of addresses can be returned:

   ```
   551-try <new@address1>
   551-try <new@address2>
   551-try <new@address3>
   551 try <new@address4>
   ```

Notice: whether a .forward file is in the home directory of a user, or whether it's in a central directory, or whether it's in a DB doesn't matter much for the design of sendmail X. Even less important is how users can edit their .forward files. sendmail X.0 will certainly not contain any program that allows users to authenticate and remotely edit their .forward file that is stored on some central server. Such a task is beyond the scope of the sendmail X design, and should be solved (in general) by some utilities that adhere to local conventions. Those utilities can be timsieved, scp, digitally signed mails to a script, updates via HTTP, etc.

2.6.7.2 Other Approaches to Aliasing?

How about the qmail approach to aliasing? Everything is just handled via one mechanism: HOME(user)/.alias[-extension]. System aliases are in HOME(aliases)/.alias[-extension]. This results in lots of file I/O which probably should be avoided.

Of course this wouldn't be flexible enough for sendmail, it must have the possibility to specify aliases via other means, e.g., maps. It might be better to put everything into maps instead of files spread out over the filesystem. In that case a program could be provided that allows a user to edit her/his own alias entry. However, such a program is certainly security critical, hence it may add a lot of work to implement properly; compare the passwd(1) command.

2.6.8 Expansion to Multiple Addresses

There have been requests to have other mechanisms than just aliases/.forward to expand an address to multiple recipients. We should consider making the AR API flexible enough to allow for this. However, there is (at least) one problem: the inheritance of ESMTP attributes, e.g., DSN parameters (see also Section 2.4.6). There are rules in RFC 1894 [Moo96a] which explain how to pass DSN requests for mailing lists and aliases. Hence for DSN parameters the rules for aliases should probably apply.

2.6.9 Virtual Hosting

It would be nice to have per-user virtual hosting. This can relieve the admin from some work. *Todo*: Compare what other MTAs offer and at least make sure the design doesn't preclude this even though it won't be in sendmail X.0. Is contrib/buildvirtuser good enough?

qmail allows to delegate virtual hosts to users via an entry in a configuration file, e.g., virthost.domain: user. Mail to address@virthost.domain goes to user-address. To keep the virtual domain use virthost.domain: user-virthost.domain. address@virthost.domain goes to user-virthost.domain-address then. Problem: lots of little files instead of a table.

2.6.10 Fallback

FallbackMXHost can be used in sendmail 8 to specify a host which is used in case delivery to other hosts fails (applies only to mailers for which MX expansion of the destination host is performed). It might be useful to make this more flexible:

- specify also a mailer to use; this may make the QMGR significantly more complicated, see also Section 3.6.3.1.
- specify under which circumstances which fallback host should be used.

The advantages/disadvantages of these proposals are not yet clear.

In theory, we could use the the second proposal to have generic bounce and defer mailers. That is, if mail delivery fails with a permanent error, the default "fallback" will be a bounce mailer, if mail delivery fails with a temporary error, the "fallback" will be a defer mailer. This would allow maximum flexibility, but the impact on the QMGR (which has to deal with all of this) is not clear.

2.6.11 Security Considerations

The address resolver should run without any privileges. It needs access to user information databases (mailbox database), but it does not need access to any restricted information, e.g., passwords or authentication data.

2.7 Initial Mail Submission

Initial mail submission poses interesting problems. There are several ways to submit e-mail all of which have different advantages and disadvantages. In the following section we briefly list some approaches.

But first we need to state our requirements (in addition to the general sendmail X requirements that have been given in Chapter 1).

- Accountability: it must be possible to identify a user who submitted an e-mail, at least on OS which have a concept of different users and require them to authenticate themselves before using the system.
- Standard conformance: The mail submission program or some subsequent cleanup process must ensure that an e-mail conforms to the appropriate standard before it is sent to its destination(s). Notice: this might be achieved by submitting e-mail to the MSA which is supposed to do cleanup anyway.

2.7.1 Initial Mail Submission Alternatives

1. Use only SMTP
   - +: no security problems
   - -: must have running SMTP daemon
   - -: error handling: mail back errors? → more complex software
2. Queue always
   - +: simple program
   - -: slow, requires "fast" queue runner or sending a notification to it (new mail in queue)
   - -: possible security problems
3. Try SMTP, queue if fails
   - +: fast (usually)
   - -: possible security problem if queued
4. How about a different delivery mechanism?

   Don't use SMTP (because it's complicated and may reject mail), but SMSP (simple mail submission protocol), submission via socket. Possible problem: how to identify other side? Always require authentication (SMTP AUTH)? Way too complicated. It's in general not possible to get the uid of the sender side, even for local (Unix socket) connections.

At the MotM (2002-08-13) option 3 was clearly favored.

See also [Ber] for more information about the problem of secure interprocess communication (for Unix).

2.7.1.1 How to use queue directory?

1. World writable directory
   (a) flat directory
      - +: easy
      - -: how to avoid DoS attacks (see postfix)?

(b) structured directory with subdirectories for each user
   - +: secure
   - -: may become large, hard to maintain

   Does this work? The files should be not world writable, so there must be some common group. Since it is not practical to have all users in the same group (and making sure that that group is used when a queue file is written), this may not work after all. Run a daemon as root, notify it of new entries: cd queuedirectory, set*uid to owner of queuedirectory, run the entry.

2. Group writable directory
   - +: "easy" to handle.
   - -: set-group-id program: security problem

2.7.1.2 Misc

Possible pitfalls: `chown` on some systems possible for non-root!

Notice: in the first version we may be able to reuse the sendmail 8 MSP. This gives us a complete MHS without coding every part.

2.7.2 Security Considerations

Since the initial mail submission program is invoked by users, it must be careful about its input. The usual measures about buffer overflows, untrusting data, parsing user input, etc. apply esp. to this program. See the Section 2.14 for some information.

Todo: Depending on the model selected above describe the possible security problems in more detail.

2.8 Mail Delivery Agents

There are several types of mail delivery agents in sendmail X similar to sendmail 8. One of them acts as SMTP client which is treated separately in Section 2.9. Another important one is the local delivery agent treated in the Section 2.8.3.

Question: does a DA (esp. SMTP client) check whether a connection is "acceptable"? Compare sendmail 8: TLS_Srv, TLS_RCPT. It could also be done by the QMGR. The DA has the TLS information, it would need to send that data to the QMGR if the latter should perform the check. That might make it simpler for the QMGR to decide whether to reuse a connection (see also Section 3.4.10.2; maybe the QMGR doesn't need this additional restriction for reuse). However, if it is a new connection it is simpler (faster) to perform that check in the DA.

2.8.1 Delivery Agent Modules

Idea: instead of having a fixed set of delivery agents and an address resolver that "knows" about all of them, maybe a more modular approach should be taken. Similar to Exim [Haz01] and Courier-MTA [Var01] delivery agents would be provided as modules which provide their own address rewriting functions. These are called in some specified order and the first which returns that it can handle the address will be selected for delivery.

sendmail 8 uses a centralized approach: all delivery agents must be specified in the .cf file and the address rewriting must select the appropriate delivery agent.

sendmail X must provide a simple way to add custom delivery agents and to select them. It seems best to hook them into the address resolver, that's the module which selects a delivery agent.

2.8.2 Specifying Delivery Agents

There must be a simple way to specify different delivery agents, i.e., their behavior and their features (see Section 3.8.2 for details). This not refers to local delivery agents (2.8.3) and SMTP clients (2.9), but also to variants of those.

In addition to specifying behavior, actual instances must be described, i.e., the number of processes and threads that are (or can be) started and are available. These two descriptions are orthogonal, i.e., they can be combined in almost any way. The configuration must reflect this, e.g., by having to (syntactical separate) structures that describe the two specifications. For practical reasons, the following approach might be feasible:

1. Declare the behavior and features of a delivery class.
2. Define actual delivery agents (instances) that implement delivery classes.

Note: sendmail 8 only specified delivery classes (called *mailers*), it does not have the need for delivery instances because it is a monolithic program that implements the mailers itself or invokes them as external programs without restrictions. In sendmail X certain restrictions are imposed, i.e., the number of processes that can run as delivery agents or the number of threads are in general limited. Even though these limits might be arbitrarily high, they must be specified.

Example:

```
delivery-class smtp { port = 25; protocol = esmtp; }
delivery-class msaclient { port = 587; protocol = esmtp; }
delivery-class lmtp { socket = lmtp.sock; protocol = lmtp; }
delivery-agent mailer1 { delivery-classes = { esmtp, lmtp };
   max_processes = 4; max_threads = 255; }
delivery-agent mailer2 { delivery-classes = { msaclient };
   max_processes = 1; max_threads = 16; }
delivery-agent mailer3 { delivery-classes = { esmtp };
   max_processes = 2; max_threads = 100; }
```

Notes:

- instead of using `delivery-agent` the actual name of the implemented module, e.g., `smtpc`, will be used in a configuration file so the module can find its configuration section.
- The example is rather contrived: all listed DAs can implement all delivery classes, the restrictions are arbitrary and make further algorithms (and implementations) more complicated.

2.8.3 Local Delivery Agent

A local delivery agent usually needs to change its user id to that of the recipient (depending on the local mail store; this is the common situation in many Unix versions). Since sendmail X must not have any set-user-id root program, a daemon is the appropriate answer to this problem (started by the supervisor, see Section 2.3).

Alternatively, a group-writable mailstore can be used as it is done in most System V based Unix systems. A unique group id must be chosen which is only used by the local delivery agent. It must not be shared with MUAs as it is done in some OSs. There is at least one problem with this approach: a user mailbox must exist before the first delivery can be performed. That requires that the mailbox is created when the user account is created and no MUA must remove the mailbox when it is empty. There could be a helper program that creates an empty mailbox for a user which however must run as root and hence will have security implications.

The local delivery agent in sendmail X will be the equivalent of `mail.local` from sendmail 8. It runs as a daemon and speaks LMTP. By default, it uses root privileges and changes its user id to that of a recipient before writing to a mailbox.

There might be other local delivery agents which use the content database access API to maximize performance, e.g., immediate delivery while the sender is waiting for confirmation to the final dot of an SMTP session.

sendmail X.0 will use a normal SMTP client – which is capable of speaking also LMTP – as an interface between `mail.local` and the queue manager. That program implements the general DA API on which the queue manager relies. The API is described in Section 3.8.4. Later versions may integrate the API into `mail.local`.

If the LDA also takes care of alias (and .forward) expansion (see Section 2.6.7.1), then sendmail X must provide a stub LDA that interfaces with custom LDAs. The stub LDA must provide the interface to the QMGR and the ability to perform .forward expansion. Its interface to the custom LDAs should be via LMTP in a first approach.

The interface to the local delivery agents must be able to provide the full address as well as just the local part (plus extensions) in all required variations. There are currently some problems with LDAs that require the full address instead of just the local part which must be solved in sendmail X. Todo: explain problems and solution(s).

2.8.4 Security Considerations

Mail delivery agents may require special privileges as explained above.

For obvious security reasons, the LDA will not deliver mail to a mailbox owned by root. There must be an alias (or some other method) that redirects mail to another account. The LDA should also not read files which require root privileges.

2.9 SMTP Client

The SMTP client is one of the mail delivery agents (see Section 2.8).

Todo: describe functionality.

Similar to SMTPS there are several architectures possible, e.g., simple (preforked) processes, multi-threaded, event-driven, or using state-threads. We need to write similar prototypes to figure out the best way to implement the SMTP clients. It isn't clear yet whether it should be the same model as SMTPS. However, it might be best to use the same model to minimize programming effort.

2.9.1 Control Flow

There are basically two different situations for a SMTPC: open a new connection (new session) or reuse an existing connection (new transaction).

New session:

1. The SMTP client receives a connection request from the QMGR (see Section 2.4) containing the required information, esp. the server address and the connection requirements. The connectiond data may include "MUST" and "SHOULD" options, e.g., try AUTH if available, must use encryption. If the connection attempt fails, inform QMGR.
2. send EHLO (or HELO), read list of features as returned from the server. Check whether required features are supported; if not, inform QMGR. The QMGR may decide to use the connection for something else (unlikely), otherwise the connection will be closed.
3. If SMTP commands are to be used that change the status of a session (e.g., STARTTLS, AUTH), those are executed and their effects are stored in the session context. The QMGR is informed if such a command fails. In some cases the SMTP session is started again, e.g., when encryption is turned on.
4. To end a session (on request of the QMGR) send QUIT.

New transaction:

1. For each transaction the envelope information and the identifier for the CDB is received from the QMGR. The SMTP client sends MAIL, RCPT, and DATA as allowed by the SMTP server (one chunk if PIPELINING is allowed, maybe up to the TCP buffer size?) and reads the replies. If a command fails, the QMGR is informed about the problem. Depending on which commands fails the entire transaction is aborted, i.e., if MAIL or DATA or all RCPT fail.
2. If the transaction is done, the QMGR is informed about the status returned by the server in response to the final dot.
3. The SMTPC sends RSET as a keep-alive check (configurable by the QMGR).

It might be useful if the data from the QMGR includes:

- timeouts after which RSET or QUIT should be used.
- timeouts for individual commands (waiting for connection open, replies to SMTP commands).
- limits on the number of recipients per transaction such that the SMTPC itselfs breaks an envelope into several transactions.

2.9.2 Security Considerations

The SMTP client will run without root privileges. It needs only access to the body of an e-mail that it is supposed to deliver. However, it may need access to authentication data, e.g., for STARTTLS: a client certificate (can be readable by everyone) and a corresponding key (must be secured), for AUTH it needs access to a pass phrase (for most mechanisms) which also must be secured[25]. For these reasons it seems appropriate that an SMTP client uses a different user id than other sendmail X programs, and achieves access to shared data (mail body, interprocess communication) via group rights.

[25] Maybe that data comes from the QMGR?

2.10 Milter

The first version of sendmail X will not support the milter API in the same way as sendmail 8 does. Any functions that allow for modifications of a mail will not be implemented. A basic milter version that supports policy decisions should be supported, however. This is necessary to implement local anti-spam features etc, especially since there is no easy extension via rulesets as in sendmail 8.

2.10.1 Possible Enhancements

The Milter API should be extended in sendmail X, even though maybe not in the first version. However, sendmail X must allow for the changes proposed here.

1. Allow milter to return a per-recipient information at the end of a transmission. This information can include rejection of recipients, rerouting to other mailers/destinations, and quarantine information.
2. Allow milter to specify ESMTP options for new recipients, e.g., DSN parameters.

Notice: if a milter is allowed to change recipient information (1) then the sendmail X architecture must allow for this. The architecture could be simpler if the address resolver solely depends on envelope information and the configuration. If it depends also on the mail content, then the address resolver must be called later during the mail transmission. This also defeats "immediate" delivery, i.e., delivery to the next hop while the mail is being received. The additional functionality will most likely be a requirement (too many people want to muck around with mail transmission). It would be nice to allow for both, i.e., mail routing solely based on envelope data, and mail routing based on all data (including mail content). There should be a configuration options which allows the MTA to speed up mail routing by selecting the first option.

2.10.2 More Ideas for Enhancements

Currently a milter can only return "continue", "accept", "temporary failure", "permanent failure", and "discard". This is less than the actions possible via access map. It might be an interesting idea to give a milter those capabilities, e.g., allow relaying or declaring a recipient as spam friend.

Question: how should access map and a policy milter interact? In sendmail 8 rejected commands are not sent to a milter at all. However, if a policy milter can return more than those few replies the interaction between access map and policy milter becomes more complicated, especially if they are conflicting. See Section 3.5.3 for further discussion.

A milter might not need the entire mail body to decide whether the mail should be rejected, hence it could be useful to have a milter specify a size limit after which the rest of the body will not be sent to the milter anymore. Note: this might not work well if the mail is in MIME format and contains multiple parts where only a later part contains something that should be rejected.

2.10.3 Security Considerations

Milters should run as a normal (unprivileged) user, but without any access to the sendmail X configuration/data files. The communication between the MTA and the milters must occur via protected means to prevent bogus milters to interfere with the operation of the MTA.

2.11 Miscellaneous Programs

2.11.1 Access to the Queue Manager

2.11.1.1 Show Mail Queue

A program (`mailq`) should be available to show the content of the mail queue(s). Various options control the output format and the kind of data shown.

2.11.1.2 Force Queue Run

It might be useful to ask the queue manager to schedule certain entries for immediate ("as soon as possible") delivery. This will also be necessary for the implementation of ETRN.

2.11.1.3 Mailstats

Some statistics need to be available. At least similar to `mailstats` in sendmail 8 and the data from the control socket. Data for SNMP should be made available. Maybe the rest is gathered from the logfile unless it can be provided in some other fashion. For example, it is probably not very useful to provide per-address statistics inside the MTA (QMGR). This would require too much resources and most people will not use that data anyway. However, it might be useful to provide hooks into the appropriate modules such that another program can collect the data in "real-time" without having to parse a logfile.

2.11.1.4 Performance Statistics

There should be some sophisticated programs that can give feedback about the performance of the MTA.

2.11.2 Security Considerations

General question: how to allow access to the data? Should we rely on the access restriction of the OS? That might be complicated since we probably have to use group access rights to share data between various modules of sendmail X. It is certainly not a good idea to give out those group rights to normal users. Moreover, some OS only allow up to 8 groups for an account. Depending on the number of modules some program has to communicate with this may cause problems.

2.12 Maps

Maps can be used to lookup data (keys, LHS) and possibly replace a key with the result of the lookup, some lookups are only used to find out whether an entry exists in the map. Ordinary maps (databases) provide only a lookup function to find an exact match[26]. There are many cases in which some form of wildcard matching needs to be provided. This can be achieved by performing multiple lookups as explained in the following.

[26] Regular expressions can be treated as maps which of course provide more functionality.

In many places we will use maps to lookup data and to replace it by the RHS of the map. Those places are: aliases, virtual hosting, mail routing, anti-spam, etc. There are some items which can be looked up in a map that need some form of wildcard matching. These are:

1. IP addresses: match full address, match subnets (omitting right-most numbers).
2. Hostnames/Domains: match full name, match subdomains.
3. E-mail addresses: match full address, match various subparts.

There's an important difference between 1 and 2: IP addresses have a fixed length[27], while hostnames can have a varying length. This influences how map entries for parts (subnets/subdomains) can be specified: for the former it is clear whether a map entry denotes a subnet while this isn't defined for the latter, i.e., `domain.tld` could be meant to apply only to `domain.tld` or also to `host.domain.tld`.

We need a consistent way to define how a match occurs. This refers to:

- Matching of subdomains: this is currently handled inconsistently. For example, for anti-relaying there is a feature `relay_hosts_only` which changes the matching from "every subdomain" to "exact".

  Possibilities:

  1. Use a leading dot to specify "only subdomains". This would require to have two entries:
     dom.ain RHS
     .dom.ain RHS
     to match the domain itself and all of its subdomains.
     So the lookup would be: full name, name without first component but with leading dot, if there is something left repeat previous step, until a lookup succeeds.
  2. Use a wildcard:
     *dom.ain RHS
     This would be more in line with the expectation of normal users due to wildcard usage in shells or regular expressions.
  3. Use an "anchor"
     @dom.ain RHS
     dom.ain RHS
     The first one will be an exact match, the second matches also all subdomains. However, this makes the lookup algorithm slightly more complicated: during the first lookup (full host name) it has to include the anchor (`@`), thereafter it must omit it. Moreover, the anchor is confusing if the entry doesn't apply to e-mail addresses but to connection information etc.

- Matching of `+detail`: How should the lookups be done in this case? And what should end up in the '%' parameters for substitution in the RHS?

  Lookup full address, Lookup address with ++ if +detail exists and detail not null, Lookup address with +* if +detail exists, Lookup address without +detail.

  Replacement:

1	user name
2	detail
3	+detail
4	omitted subdomain?

[27] in IPv6 subparts can be omitted however.

As usual localparts are anchored with a trailing `@` to avoid confusion with domain names.

Notice: the "detail" delimiter should be configurable. sendmail 8 uses `+`, other MTAs use `-`. This may interfere with aliases, e.g., owner-list. Question: how to solve this problem?

2.13 Modules

As explained in Section 1.1.3.4, sendmail X must provide hooks for extensions. One possible way are modules, similar to Apache [ASF]. Modules help to avoid having large binaries that include everything that could ever be used.

2.13.1 Security Considerations

Modules must only be loaded from secure locations. They must be owned by a trusted user.

2.14 Security Hints

This section contains a hints and thought on how to design and implement programs, esp. those related to MTAs, to ensure they are secure.

1. Do as little as possible in set-group-ID/set-user-ID programs or completely avoid them.

   It has not yet been decided whether the initial mail submission program (see Section 2.7) will be set-group-ID. No program in sendmail X is set-user-ID root.

   A set-group-ID/set-user-ID program must operate in a very dangerous environment that can be controlled by a malicious user. Moreover, the items that must be checked varies from OS to OS, so it is difficult to write portable code that cleans up properly.

2. Do as little as possible as root.

   Only use root if absolutely necessary. Do not keep root privileges because they might be needed later on again, consider splitting up the program instead.

   Avoid writing to root owned files. Question: is there any situation where this would be required? Avoid reading from files that are only accessible for root. This should only be necessary for the supervisor, since this program runs as root so its configuration file should be owned by root. Otherwise root would have to rely on the security of another account.

3. Programs and files are not addresses. Don't treat them as such.

   sendmail 8 treats programs and files as addresses. Obviously random people can't be allowed to execute arbitrary programs or write to arbitrary files, so sendmail 8 goes through contortions trying to keep track of whether a local user was "responsible" for an address. This must be avoided.

   The local delivery agent can run programs or write to files as directed by `$HOME/.forward`, but it must always run as that user. The notion of "user" should be configurable, but root must never be a user. To prevent stupid mistakes, the LDA must make sure that neither `$HOME` nor `$HOME/.forward` are group-writable or world-writable.

   Security impact: Having the ability to write to `.forward`, like `.cshrc` and `.exrc` and various other files, means that anyone who can write arbitrary files as a user can execute arbitrary programs as that user.

4. Move separate functions into mutually untrusting programs. However, make sure you don't go overboard here, use a reasonable separation.

5. Don't trust data or commands from untrusted sources or programs. Check for correct syntax and valid data as well as resource consumption.

   Do not assume that data written to disk is secure. If at all possible, assume that someone may have altered it. Hence no security relevant actions should be based on it.

6. Don't parse.

   The essence of user interfaces is *parsing*: converting an unstructured sequence of commands into structured data. When another program wants to talk to a user interface, it has to *quote*: convert the structured data into an unstructured sequence of commands that the parser hopefully will convert back into the original structured data. This situation calls for disaster. The parser often has bugs: it fails to handle some inputs according to the documented interface. The quoter often has bugs: it produces outputs that do not have the right meaning. When the original data is controlled by a malicious user, many of these bugs translate into security holes (e.g., `find | xargs rm`).

   For e-mail, only a few interfaces need parsing, e.g., RFC 2821 [Kle01] (SMTP) and RFC 2822 [Res01] (for mail submission). All the complexity of parsing RFC 2822 address lists and rewriting headers must be in a program which runs without privileges.

7. Keep it simple, stupid.

   Security holes can't show up in features that don't exist. That doesn't mean that sendmail X will have almost no features, but we have to be very carefull about selecting them and their security and reliability impact.

   Especially the availability of several options can cause problems if a program can access data that is not directly accessible to the user who calls it. This applies not only to set-group/user-ID programs, but also daemons that answer requests. This has been demonstrated by sendmail 8, e.g., debug flags for queue runners, which reveal private data.

8. C doesn't have boundary checks. Be careful.

   C "strings" are inherently dangerous. Use something else which prevents buffer overflows.

9. Do not rely on one layer of security (defense). If a program does something security relevant, try to use several layers of defense against attacks. If one is broken, another one hopefully kicks in.

There are more things to security than just these programming advices. For example, a program should not leak privileged (private/confidential) information. This applies to data that is logged or made available via debugging options. A program must also prevent being abused to access data that it can read due to its rights and being tricked into making that data available to an attacker, neither for reading nor writing.

2.14.1 Privileged Access

Question: where does sendmail need privileged access? The following sections provide a list and hopefully solutions.

2.14.1.1 Access to Files

- :include: files (reading); handle like .forward files
- Possible solutions for the .forward problem:
  - expand in LDA: LDA can return redirect addresses.
  - special map database, `forward -e` similar to passwd(1).
  - special directory /var/forward
  - cron tool to migrate HOME/.forward
- SMTP AUTH in server: access to password database: can be done via a daemon
- Access to maps:

  LDAP with Kerberos: there should be a way to do this without root privileges. This might be a documentation issue (kinit before starting sendmail, chown ticket file)

  PH map: it's possible... (how?)

2.14.1.2 Sockets

- bind(2) to reserved port in server.

  Problem: the server may close(2) the socket due to errors or load conditions, e.g., RefuseLA, MaxChildren in sendmail 8. In that case the server needs to bind(2) to the port again later on. Since the server is not supposed to run with root privileges, another program (the MCP) must take care of that, i.e., it is notified of the problem and can either start a new server or pass an open fd to the server.

  Note: To bind(2) to a reserved port may not require root on all OS variants, there might be other access control methods, e.g., a different, privileged user id that is allowed to bind(2) to certain ports.

  Alternatively, a server might not close(2) the socket, Instead of close(2), accept(2) and give 421 error, then close(2) just the connection.

  Side note: RefuseLA, MaxDaemonChildren should be configurable per DaemonPortOption.
- bind(2) to reserved port (ClientPortOptions Port=) for client. Do not support this in sendmail X, it's a bad idea anyway.

2.14.1.3 Running as a different user

- Delivery to files (setreuid()) (mailfile())
  - From .forward/:include:/alias files
  - Handle like delivery to programs
- Delivery to programs (fork, setuid()) (deliver())
  - LDA delivery (F=S, U=uid)
  - From .forward/:include:/alias files
  - LDA expands forward files

  - extend LMTP to return forward redirects to the MTA to save re-invocation
  - What to do if user delivers to two programs and one tempfails?

- program map: runs as address resolve user.

2.15 Misc

2.15.1 Misc Misc

Todo: structure this.

Can we keep interfaces abstract and simple enough so we can use RPCs[28]? This would allow us to build a distributed system. However, this must be a compile time option, so we can "put together" an MTA according to our requirements (all in one; some via shared libraries; some via sockets; some via RPCs). See also Section 3.1.1.

Rulesets, esp. check_*: make the order in which things happen flexible. Currently it's fixed in proto.m4, which causes problems; (tempfail in parts: require rewrite, even then it's hard to maintain etc). Use subroutines and make the order configurable (within limits).

Use of mode bits to indicate status of file? e.g., for forward: it +t: being edited right now, don't use (temp.fail.) for queue files: +x completely written.

Can several processes listen on a socket? Yes, but there is a "thundering herd" problem: all processes are woken up, but only one gets the connection. That is inefficient for large number of processes. However, it can be mitigated by putting locks around the file such that only one process will do an `accept()`. See [Ste98] for examples.

2.15.2 Configuration

Configuration: instead of having global configuration options why not have configuration functions? For example: Timeout.Queuereturn could be a function with user-defined input parameters (in form of macros?):

Timeout.Queuereturn(size, priority, destination) = some expression.

This way we don't have to specify the dependency of options on parameters, but the user can do it. Is this worthwhile and feasible? What about the expression? Would it be a ruleset? Too ugly and not really useful for options (only for addresses). Example where this is useful: the FFR in 8.12 when milters are used (per Daemon). Example where this is already implemented: srv_features in 8.12 allows something like this.

Problem: which macros are valid for options? For example, recipient can be a list for most mails.

2.15.3 Configuration Changes

Configuration changes may cause problems because some stored data refers to a part of the configuration that is changed or removed. For example, if there are several retry schedules and an entry in the queue refers to one which is removed during a configuration change, what should we do? Or if the retry schedule

[28]Remote Procedure Calls, not referrring to the actual implementation by Sun

is changed, should it affect "old" entries? sendmail 8 pretty much treats a message as new every time it processes it, i.e., it dynamically determines the actual delivery agent, routing information, etc. This probably can solve the problem of configuration changes, but it is certainly not efficient. We could invalidate stored information if the configuration changes (see also Section 3.13.6).

2.15.4 Performance Measurements: Profiling

Most sendmail X programs must have a compilation switch to turn on profiling (not just -pg in the compiler). Such a switch will turn on code (and data structures) that collect statistics related to performance. For example, usage (size, hit rate) of caches, symbol tables, general memory usage, maybe locking contentions, etc. More useful data can probably be gathered with getrusage(2). However, this system call may not return really useful data on most OS. On OpenBSD 2.8:

```
long ru_maxrss;          /* max resident set size */
long ru_ixrss;           /* integral shared text memory size */
long ru_idrss;           /* integral unshared data size */
long ru_isrss;           /* integral unshared stack size */
```

seem to be useless (i.e., 0). SunOS 5.7: NOTES: Only the timeval member of struct rusage are supported in this implementation.

A program like top might help, but that's extremely OS dependent. Unless we can just link a library call in, we probably don't want to use this.

2.15.5 Logging

There are various requirements for logging:

- selection of what is being logged must be flexible (see Section 2.15.5.1)
- logfiles must be easy to parse and analyze (see Section 2.15.5.2)
- the logging mechanism must be extensible, i.e., it must be possible to log to a database, a custom application, etc. (see Section 2.15.5.3)

Note: a different approach to logging is to use the normal I/O (well, only O) operations and have a file type that specifies logging. The basic functionality for that is available in the sendmail 8/9 I/O layer. However, it seems that this approach does not fulfill the requirements that are stated in the following sections.

2.15.5.1 Logging Granularity

The logging in sendmail X must be more flexible than it was in older versions. There are two different issues:

1. logging levels
2. logging per functionality

About item 1: the current version (sendmail 8) uses: LogLevel and the syslog priorities (LOG_EMERG, LOG_ERR, ...). The latter can be used to configure where and how to log entries via syslog.conf(5). The loglevel can be set by the administrator to select how much should be logged. Note: in some sense these are overlapping: syslog priorities and loglevels are both indicators of how important an log event is. However, the former is not very fine grained: there are only 8 priorities, while sendmail allows for up to 100 loglevels. Question: is it useful to combine both into a single level or should they be kept separate? If they are kept separate, is there some correlation between them? For example, it doesn't make sense to log an error with priority LOG_ERR but only if LogLevel is at least 50. ISC [ISC01] combines those into a single value, but it basically uses only the default syslog priorities and then additionally debug levels.

An orthogonal problem (item 2) is logging per "functionality"[29]. There are many cases where it is useful to select logging granularity dependent on functionalities provided by a system. This is similar to the debugging levels in sendmail 8. So we can assign a number to a functionality and then have a LogLevel per functionality. For example, -L8.16 -L20.4 would set the LogLevel for functionality 8 to 16 and for 20 to 4. Whether we use numbers or names is open for discussion.

syslog offers facilities (LOG_AUTH, LOG_MAIL, ..., LOG_LOCALx), however, the facility for logging is fixed during the openlog() call, it is not a parameter for each syslog() call. This is serious drawback and makes the facility fairly useless for software packages that consists of several parts within a single process like sendmail 8 (which performs authentication calls, mail operations, and acts as daemon (at least)).

ISC ([ISC01], see Section 3.16.16.1) offers categories and modules to distinguish between various invocations of the logging call. Logging parameters are per category, i.e., it is possible to configure how entries are logged per category and per priority. The category is similar to the syslog facility but it is an argument for each logging call and hence offers more flexibility. However, ISC does not offer loglevels beyond the priorities. A simple extension can associate loglevels with categories and modules. If the loglevel specified in the logging call is larger than the selected value, then the entry will not be logged.

Misc: should we use log number consisting of 16 bit category and 16 bit type number?

2.15.5.2 Parsing Logfiles

Logfiles must be easy to parse and analyze. For parsing it is very helpful if simple text tools like awk, sed, et.al. can be used instead of requiring a full parser, e.g., one that understands quoting and escaping.

The basic structure of logfile entries is a list of fields which consist of a name and a value, e.g.,

```
name1=value1, name2=value2, ...
```

The problems here are whether the delimiters (space or comma) are unique, i.e., whether they do not appear in the name or the value of a field. While this is easy to guarantuee for the name (because it's chosen by the program), values may contain those delimiters because they can be (indirectly) supplied by users. There are two approaches to solve this problem:

1. use a unique delimiter that cannot appear in a value;
2. change the representation of values such that the chosen delimiters are unique.

If we would relax the requirement to use simple text tools we could use a different representation, e.g., RCBs as described in Section 3.16.11.1.

[29] also called category by ISC or facility by syslog

Proposal 1 is not easy to achieve since values are user controlled as explained before. Approach 2 seems to be more promising, even now there is some encoding happening in sendmail 8, i.e., non-printable characters are replaced by their octal representation (or in some cases simply by another character, e.g., '?'). A simple encoding scheme would be: replace space with underscore, escape underscore and backslash by a leading backslash. The decoding for this requires parsing to see whether underscores or backslashes where escaped. This encoding allows to use space as delimiter[30]. A different proposal is not to use spaces as delimiters (and hence not to change them), but commas or another fairly unique character. Which character (besides the obvious ',' and ';') would be a "good" delimiter, i.e., would not commonly appear in a value?

2.15.5.3 Extensible Logging

The logging functionality must be abstracted out, i.e., we have to come up with a useful API and provide one library for it, which would use syslog(). Other people can replace that library with their own, e.g., for logging into a database, files, or whatever.

2.15.6 Debugging

A simple level of debugging can be achieved by turning on verbose logging. This may be via additional logging options, e.g., -dX.Y, similar to sendmail 8, but the output is logged not printed on stdout.

Additionally, there must be some way to start a program under a debugger. Remember: most programs are started by the supervisor, so it is not as simple as in sendmail 8 to debug a particular module. Take a look at postfix for a possible solution.

2.15.7 Robust Programming

sendmail X must behave well in the case of resource shortages. Even if memory allocation fails, the program should not just abort, but act in a fail-safe manner. For example, a program that must always run even in the worst circumstances is the queue manager. If it can't allocate memory for some necessary operation, it must fall back to a "survival" mode in which it does only absolutely necessary things and shuts down everything else and then "wait for better times" (when the resource shortage is over). This might be accomplished by reserving some memory at startup which will be used in that "survival" mode.

postfix components just terminate when they run out of memory if a `my*alloc()` routine is used. This is certainly not acceptable by some parts of postfix nor sendmail X. Library routines especially shouldn't have this behavior.

2.16 Schedule

sendmail X will be developed in several stages such that we have relatively soon something to test and to experiment with.

First, the design and architecture must be specified almost completely. Even if not every detail is specified, every aspect of the complete system must be mentioned and be considered in the overall design. We don't want to patch in later new parts which may require a redesign of some components, esp. if we already worked on those.

[30] of course the encoding can be changed if a different delimiter should be used

However, the implementation can be done in stages as follows: the first version will consist only of a relaying MTA, i.e., a system that can accept mail via SMTP and relay it to delivery agents that speak SMTP or LMTP. This way we will have an almost complete system with basic functionality. The "only" parts that are missing are other delivery agents, an MSP, and some header rewriting routines etc.

2.17 Glossary

This section contains the explanation of terms used in this (and related) documents. The terms are usually taken from various RFCs.

- Mail Body: part of mail content, may be structured according to MIME (RFC 2045) if so specified in the headers.
- Mail Content: part of mail object after DATA: consists of headers and body (RFC 2821).
- Mail Data: same as Mail Content.
- Mail Envelope: Consists of one sender and one or more recipients (RFC 2821).
- Mail Headers: Collection of field/value parts (RFC 2822), must be US ASCII.
- Mail Object: contains envelope and content. SMTP transports mail objects (RFC 2821).
- Resolved address: a tuple that contains the delivery agent, the host(s) to which to connect, and the address to present to those hosts as recipient. See Section 3.6.3.1 for details.
- SMTP Session: see RFC 2821.
- SMTP Transaction: see RFC 2821.

Chapter 3

Sendmail X: Functional Specification

3.1 Functional Specification of sendmail X

This chapter describes the external functionality as well as the internal interfaces of sendmail X as much as required, i.e., for an implementation by experienced software engineers with some knowledge about MTAs etc. In each section it will be made clear what is the external functionality and which API or other interface (in the form of a protocol) will be provided.

The internal functionality describes the interfaces between the modules of sendmail X. As such they must never be used by anything else but the programs of which sendmail X consists. These interfaces are subject to changes without notice. It is not expected that modules from different versions work with each other, even though it might be possible.

An external interface is one which is accessible by a user.

3.1.1 Asynchronous APIs

Many of the sendmail X modules require that function calls do not block. If a function can block then an asynchronous API is required. Unfortunately it seems hard to specify an API that works well with blocking and non-blocking calls. Hence the APIs will be specified such that most (or all) blocking functions are split in two: one to initiate a function (the *initiate* function) and another one (the *result* function) to get the result(s) of the function call. To allow for a synchronous implementation, the first (initiate) function should also provide result parameters and a status return. The status indicates whether the result parameters have valid values, or whether the result will be returned later on via the second (get result) function. This convention seems to make it possible to provide APIs that work for both (synchronous and asynchronous) kind of implementations.

The calling program should not be required to poll for a result, but instead it should be notified that a result is available. This can be done via some interprocess notification mechanism, e.g., System V message passing or via a message over a socket through which both programs communicate. Then those programs can use `select(2)`, `poll(2)`, or something similar to determine whether a message is available. Note: this requires that both programs agree on a common notification mechanism which can be a problem in case of (third-party) libraries. For example, OpenLDAP [LDA] does provide an asynchronous interface (e.g., `ldap_search(3)`), but it does not provide an "official" notification mechanism[1], it requires polling

[1] accessing an internal file descriptor allows to get notified when a result arrives, but is not a documented API

(`ldap_result(3)`). It should be possible to have a function associated with a message (type) such that the main loop is just a function dispatcher (compare engine in libmilter).

3.1.1.1 Generic Description of Asynchronous Operation

In this section we describe how asynchronous "function" calls can be handled. For the following discussion we assume a caller *C* and a callee *S*. The caller *C* creates a request *RQ* that contains the arguments for a function *F*, and a token *T* which uniquely identifies the request *RQ* (we could write *RQ(T)*). Note: it is also possible to let *S* generate the token if the argument is a result parameter (pointer to a token which can be populated by *S*). Additionally, the argument list may contain a result parameter *PR* in which a result can be immediately stored if the call can be handled synchronously. *C* enqueues the fact that it made the function call by storing data describing the current state *St(T)* in a queue/database DB along with the identifier *T*. The request is sent to *S* (may be appended to a queue for sending via a single communication task instead of sending it directly) unless it can be handled immediately and returned via the result parameter *PR* (the return status must specify that the result has been returned synchronously). A communication task receives answers *A* from *S*. The answer contains the identifier *T* which is used to lookup the current data *St(T)* in the DB. Then a function (the *result* or *callback* function `RC`) is invoked with *St* and *A* as parameters. It decodes the answer *A* (which contains the function result) and acts accordingly, also taking the current state *St* into account and probably modifying it accordingly. Additionally, it may use the result parameter *PR* to store the result in the proper location. The result parameter may provide some buffer (or allocated) structure in which to store the result such that the callee is not responsible for the allocation. However, the details of this depend on the functions and their requirements, e.g., their output values (it may be just a scalar).

To handle immediate status returns, the function that sends (enqueues) the request has to be able to deal with that. Such an implementation is an enhancement for a later version (probably not for 9.0).

3.1.1.2 Some Remarks about Identifiers

There are several kinds of identifiers that are used within sendmail X for various purposes. Identifiers (handles, tags) in general must of course be unique during the time in which they are used to identify an object. Beyond this obvious purpose, some identifiers should fulfill other requirements.

1. Some identifiers are supposed to be unique for a long time, e.g., for sessions and transactions. It seems useful to make these unique for a much longer time which allows them to be used as identifiers in logfiles over a long period. These identifiers should also be easily representable in human-readable form.

2. Identifiers for asynchronous operations (see Section 3.1.1.1) are very short-lived, they need to be only valid while the transaction is not yet finished. In some cases it might be useful to assure that they are unique even after the end of a transaction to avoid problem with "hanging" transactions (compare TCP/IP?).

   If a handle is only used by the caller to identify an object in its memory, then it seems a good idea to use the memory address of that object as handle (as for example done by aio(3), see `aio_return(3)`). The advantages are:

   - It doesn't require any lookup functions that map the handle to the address of the object.
   - The handles are short (not 20 bytes like the current session identifiers).
   - No function is required to generate them.

The disadvantages are:

- 32/64bit address pointers must be handled appropriately. This could be configured at compile time and "encoded" in the protocol version.
- The objects must not move. Notice: this may happen if an object is moved from one DB to another one while a request is outstanding.
- The handle is not unique over a longer time period.

3.2 Configuration

3.2.1 Configuration Structure

Section 2.2.5 gives an example how two SMTP servers are defined and reference common parts (aliases). However, there are some problems due to the modularized implementation of the MTS: many operations are performed by the address resolver, e.g., access map lookups and routing decisions including whether a recipient address is local. It is easy to imagine a configuration in which different servers listen on different IP addresses and need different configurations, e.g., a map of local users or even anti-spam configurations. Most MTAs only offer virtual hosting, i.e., one MTA is responsible for several domains, but those domain have basically the same configuration. sendmail 8 has some options per "daemon" (`DaemonPortOptions`) and several requests have been made to have more options "per daemon". This is easy in sendmail X as long as those options are "local" to the server, but it is complicated when other modules provide the functionality.

There are two problems involved here:

1. options are currently specified in the module which implements the functionality, e.g.,
   - access map is specified in smar and a flag in smtps turns its usage one.
   - a mailertable is specified in smar and used by smtps to decide whether a recipient is local and by qmgr for routing addresses.
2. some functionality provided by an option is used by different modules, e.g., mailertable (as explained above).

Problem 1 could be solved as explained in Section 2.2.5.

Problem 2 could be solved by separating the functionality, e.g., mailertable could be split into two tables: one to decide where a recipient is local, and another one for routing addresses. However, this may increase maintainance costs as some information is duplicated in different tables which may cause consistency problems.

3.2.2 Naming Conventions

It is important to use consistent naming; not just in the source code but especially in the configuration file.

In the following some proposals are given to achieve this. First about the structure of entries:

1. Entries can consist of components which are delimited by underscores (or dots).

2. Entries are named in a *top-down* manner: the most *important* component is first, e.g., `queue_timeout`. However, if an *adjective* is used, it may make sense to use the "natural" ordering, i.e., the adjective before the noun, e.g., `max_processes`. These two rules may be used within a single entry, e.g., `smtps_max_connections`.

3. Whenever useful, entries should be structured, not written as components. This is similar to breaking a program into several functions instead of inlining everything. However, there is no clear rule about this. Some rules of thumb:

   (a) If there are several entries (more than three(?)) with the same *prefix*, then creating a structured entry is useful.

   (b) If there are several entries (more than two(?)) with the same *prefix* which are used in different components, then creating a structured entry is useful.

4. Abbreviations are used sparringly and if they are used, then they are used everywhere, they should not be mixed with the unabbreviated words.

Next about the (components of) names used:

1. `entries`: number of entries in some structure.

2. `size`: size of some structure (in KB, MB, etc)

3. `space`: required free space for some structure in some other medium, e.g., on disk.

4. `level`: level at which some events should occur, usually refers to producing some kind of output, e.g., `log_level`.

3.2.3 Specification of Classes (Lists, Sets)

Question: how should *classes* (lists, sets) be specified? Should it be just a lists of entries in the configuration file? sendmail 8 allows to load classes from other sources, e.g., from an external file, a program, or a map lookup. In the latter case a key is looked up in a map and the RHS (split?) is added to the specified class. It might be useful to have also a map which simply specifies the valid members as keys, i.e., lookup a value and if it is found, it is a member of the class for which the map is given. More complicated would be that the RHS contains a list of class names which would be compared against the class that is checked.

3.3 Supervisor

3.3.1 External Interfaces

The supervisor doesn't have an external interface, except for starting/stopping it. Reconfiguring is probably done via stop/start, or maybe via sending it a HUP signal. The latter would be good enough to enable/disable parts of sendmail X, i.e., edit the configuration file and restart the supervisor. In the first version, there will be no special commands to enable/disable parts of sendmail X.

The supervisor can be configured in different ways for various situations. Example: on a gateway system local delivery is not required hence the LDA daemon is not configured (or started).

3.3.2 Operation

The MCP starts processes on demand, which can come from several sources:

1. The configuration requires more than zero processes (see Section 3.3.4).
2. There is a communication request coming in on the socket on which the process is supposed to listen, but there is no process available (and the maximum number of processes has not been reached yet).
3. Another process requests that a process is started, a typical example would be the QMGR asking for more SMTPS or DA processes.

The MCP keeps track of the number of processes and acts accordingly, i.e., when the upper limit is reached no further processes will be started, when a lower limit is reached more processes are started. Child processes also inform the MCP whenever they finished a task and are available for a new one.

Case 2: If there is no process available then the MCP listens on the communication channel and starts a process when a communication comes in. In that case the connection will be passed to the new process (in the form of an open file descriptor). This is similar to inetd or postfix's master program. Question: how to deal with the case where processes are available? Should those processes just call accept() on the connection socket? That's the solution that the postfix master program uses; the child processes have access to the connection socket and all(?) of them listen to it (Wietse claims the thundering herd problem isn't really a problem since the number of processes is small).

Question: would it be more secure if only a program can request to start more copies of itself? However, that would probably increase the program complexity (slightly) and communication overhead. Decision: designated programs can request to start other programs.

Question: should the MCP use only one file descriptor to receive requests from all processes it started or should it have one fd per process or process type? The latter seems better since it allows for a clean separation, even though it may require (a few) more fds.

3.3.3 Shutdown

There are several types of shutdown:

- "normal" shutdown: the programs are told to shutdown without interrupting ongoing work. However, no new work (connections etc) is started anymore.
- immediate shutdown: the programs must shut down immediately, ongoing work must be terminated, e.g., open connections are closed.
- crash: the system or some programs crash without having the chance for an orderly shutdown. If this happens, the state of the system must be recoverable, i.e., there must be no inconsistency after startup, or only those which can be resolved without loss of accepted mails.

Question: how to distinguish between immediate shutdown and "normal" shutdown? The former is probably triggered by a signal (SIGTERM), the latter by a command from the MCP (or some other control program).

The supervisor is responsible for shutting down the entire sendmail X system. To achieve this, the receiving processes are first told to stop accepting new connections. Current connections are either terminated (if immediate shutdown is required, e.g., system will be turned off fast) or are allowed to proceed (up to a certain amount of time). The queue manager will not schedule any delivery attempts any more and wait for outstanding connections to terminate. Delivery agents will be told to terminate (again: either immediately or orderly). The helper programs, e.g., address resolver, will be terminated as soon as the programs which require them have stopped.

3.3.4 Configuration

The configuration of the MCP might be similar to the `master` process in postfix. It contains a list of processes, their types, the uid/gid they should run as, the number of processes that should be available, how they are supposed to be started, etc. It should also list how the processes communicate with the MCP (fork()/wait(), pipe), how often/fast a process can be restarted before it is considered to fail permanently. The latter functionality should probably be similar to (x)inetd.

Here's the example master.cf[2] file from postfix:

> Postfix master process configuration file. Each line describes how a mailer component program should be run. The fields that make up each line are described below. A "-" field value requests that a default value be used for that field.
>
> Service: any name that is valid for the specified transport type (the next field). With INET transports, a service is specified as host:port. The host part (and colon) may be omitted. Either host or port may be given in symbolic form or in numeric form. Examples for the SMTP server: localhost:smtp receives mail via the loopback interface only; 10025 receives mail on port 10025.
>
> Transport type: "inet" for Internet sockets, "unix" for UNIX-domain sockets, "fifo" for named pipes.
>
> Private: whether or not access is restricted to the mail system. Default is private service. Internet (inet) sockets can't be private.
>
> Unprivileged: whether the service runs with root privileges or as the owner of the Postfix system (the owner name is controlled by the mail_owner configuration variable in the main.cf file).
>
> Chroot: whether or not the service runs chrooted to the mail queue directory (pathname is controlled by the queue_directory configuration variable in the main.cf file). Presently, all Postfix daemons can run chrooted, except for the pipe and local daemons. The files in the examples/chroot-setup subdirectory describe how to set up a Postfix chroot environment for your type of machine.
>
> Wakeup time: automatically wake up the named service after the specified number of seconds. A ? at the end of the wakeup time field requests that wake up events be sent only to services that are actually being used. Specify 0 for no wakeup. Presently, only the pickup, queue manager and flush daemons need a wakeup timer.
>
> Max procs: the maximum number of processes that may execute this service simultaneously. Default is to use a globally configurable limit (the default_process_limit configuration parameter in main.cf). Specify 0 for no process count limit.
>
> Command + args: the command to be executed. The command name is relative to the Postfix program directory (pathname is controlled by the program_directory configuration variable).

[2]this file can be changed by the user within the restriction listed in caps inside the file. Wietse is certainly tired of people breaking postfix by not following the instructions.

Adding one or more -v options turns on verbose logging for that service; adding a -D option enables symbolic debugging (see the debugger_command variable in the main.cf configuration file). See individual command man pages for specific command-line options, if any.

SPECIFY ONLY PROGRAMS THAT ARE WRITTEN TO RUN AS POSTFIX DAEMONS. ALL DAEMONS SPECIFIED HERE MUST SPEAK A POSTFIX-INTERNAL PROTOCOL.

DO NOT CHANGE THE ZERO PROCESS LIMIT FOR CLEANUP/BOUNCE/DEFER OR POSTFIX WILL BECOME STUCK UP UNDER HEAVY LOAD

DO NOT CHANGE THE ONE PROCESS LIMIT FOR PICKUP/QMGR OR POSTFIX WILL DELIVER MAIL MULTIPLE TIMES.

DO NOT SHARE THE POSTFIX QUEUE BETWEEN MULTIPLE POSTFIX INSTANCES.

# service #	type	private (yes)	unpriv (yes)	chroot (yes)	wakeup (never)	maxproc (50)	command + args
smtp	inet	n	-	n	-	-	smtpd
#628	inet	n	-	n	-	-	qmqpd
pickup	fifo	n	n	n	60	1	pickup
cleanup	unix	-	-	n	-	0	cleanup
qmgr	fifo	n	-	n	300	1	qmgr
#qmgr	fifo	n	-	n	300	1	nqmgr
rewrite	unix	-	-	n	-	-	trivial-rewrite
bounce	unix	-	-	n	-	0	bounce
defer	unix	-	-	n	-	0	bounce
flush	unix	-	-	n	1000?	0	flush
smtp	unix	-	-	n	-	-	smtp
showq	unix	n	-	n	-	-	showq
error	unix	-	-	n	-	-	error
local	unix	-	n	n	-	-	local
virtual	unix	-	n	n	-	-	virtual
lmtp	unix	-	-	n	-	-	lmtp

Interfaces to non-Postfix software. Be sure to examine the manual pages of the non-Postfix software to find out what options it wants. The Cyrus deliver program has changed incompatibly.

```
cyrus   unix -n n --pipe
  flags=R user=cyrus argv=/cyrus/bin/deliver -e -m ${extension} ${user}
uucp   unix -n n --pipe
  flags=Fqhu user=uucp argv=uux -r -n -z -a$sender - $nexthop!rmail ($recipient)
ifmail    unix  -       n       n       -       -       pipe
  flags=F user=ftn argv=/usr/lib/ifmail/ifmail -r $nexthop ($recipient)
bsmtp     unix  -       n       n       -       -       pipe
  flags=Fq. user=foo argv=/usr/local/sbin/bsmtp -f $sender $nexthop $recipient
```

First take at necessary configuration options for MCP:

- name of service (used as identifier).
- connection type: inet, unix, fifo. For inet: specify also address:port (address can be omitted), for unix: path name (with appropriate default).

- private: connection will not be accessible to non-sendmail X programs.
- user/group: which user to run as.
- chroot: use chroot (which directory?).
- feedback type: just termination information, or allowed to request start/termination of other programs.
- wakeup: wake up process at certain intervals (Question: is this necessary? Probably not.)
- minimum and maximum number of processes.
- number of crashes which are allowed per time unit before the service is shut down permanently.
- grouping: it might be necessary to shut down other programs if one crashes due to their interdependencies, e.g., if they access a Berkeley DB.
- ordering: some processes must be started or even be running before others.
- command and arguments.

Notice: using a syntax as in master.cf or inetd.conf is not a good idea, since it violates the requirements we listed in Section 2.2.2. The syntax must be the same as for the other configuration files for consistency.

3.3.5 Internal Interfaces

The supervisor starts and controls several processes. As such, it has a control connection with them. In the simplest case these are just `fork()`, `wait()` system calls, in more elaborate case it may be a socket over which status and control commands are exchanged.

The supervisor may set up data (fd, shared memory, etc) that should be shared by different processes of the sendmail X system. Notice: since the MCP runs as root, it could setup sockets in protected directories to which normal users don't have access. Open file descriptors to those sockets (for communication between modules) could then be passed on to forked programs. This may help to increase the security of sendmail X due to the extra protection. However, it requires that the MCP sets up all the necessary communication files. Moreover, if a program closes the socket (as SMTPS may do for port 25) it can't reopen it anymore and hence must get the file descriptors from the MCP again (either by passing a fd or by terminating and being started). This may not be really useful, but it's just an idea to be noted.

The MCP will bind to port 25 as explained earlier (see Section 2.3) and hand the open socket over to the SMTPS (after changing the uid).

3.4 Queue Manager

Question: Will this be a (purely) event driven program?

Question: worker model or thread per functionality? It won't be a single process which uses event based programming since this doesn't scale on multi-processor machines. It seems a worker model is more appropriate: it is more general and we might be able to reuse it for other modules, see also Section 3.20.2.

3.4.1 External Interfaces

The queue manager doesn't have an external interface. However, it can be configured in different ways for various situations. *Todo:* and these are?

Such configuration options influence the behavior of the scheduler, the location of the queues, the size of memory caches, etc.

3.4.2 Shutdown

The queue manager will not schedule any delivery attempts any more. It will wait for outstanding connections to terminate unless an immediate shutdown is requested. The incoming queue will be flushed to the deferred queue. Delivery agents are informed by the MCP to stop. The QMGR is waiting for them to terminate and records the delivery status in the deferred EDB.

3.4.3 Internal Interfaces

The status of the queue manager must be accessible from helper programs, e.g., see Section 2.11.1 The QMGR does not provide this as user accessible interface to allow for changing the internal protocols without having to change programs outside sendmail X.

Note: due to the complexity of the QMGR the following sections are not structured as subsection of this one because the nesting becomes too deep otherwise.

3.4.4 Indices for Accessing the EDBs

The main index to access the deferred EDB will be the time of the next try. However, it is certainly useful to send entries to a host for which a DA has an open connection. Question: what do we use as index to find entries in an EDB to reuse an open connection? An EDB stores canonified addresses or resolved addresses (DA, host, address), it does not store MX records. Those are kept in a different cache (mapping), if they are cached at all. We cannot use the host signature for lookups as sendmail 8 does for piggybacking for the same reason: the host signature consists of the MX records. 8.12 uses an enhanced version for piggybacking, i.e., not the entire host signatures need to match, but only the first. Maybe it is sufficient for selection of entries from an EDB to use the host name (beside all the other criteria as listed in Section 2.4.4.1). The actual decision to reuse a connection is made later on (by the micro scheduler, see Section 2.4.4.3). That decision is based on the host name/IP address and maybe the connection properties, e.g., STARTTLS, AUTH data.

If a DA has an open connection, then that data is added to the outgoing open connection cache (an incoming connection from a host may be taken as hint that the host can also receive mail). The hint is given to the schedulers, which may change their selection strategy accordingly. The first level scheduler can reverse map the host name/IP address to a list of destination hosts that can appear in resolved addresses. Then a scan for those hosts can be made and the entries may be moved upward in the queue.

Question: do we only lookup queue entries for the best MX record (to piggyback other recipients on an open connection)? We could lookup queue entries for lower priority MX records if we know the higher ones are unavailable. It may be a question of available resources (computational as well as network I/O) whether it is useful to perform such fairly complicated searches (and decision processes) to reuse a connection. In some cases it might be simpler (and faster) to just open another connection. However, it might be "expensive" to setup a connection, esp. if STARTTLS or something similar is used. A really sophisticated scheduler may take this into account for the decision whether to use an existing connection or whether to open new ones. For example, if the number of open connections is large then it is most likely better to reuse an existing connection. Those numbers (total and per domain) will be configurable (and may also depend on the OS resources).

3.4.5 Interface to SMTP DAs

Question: which host information does the QMGR send to the DA? Only the host from the resolved address tuple (so the DA does MX lookups), the MX list, only one host out of the MX list, or only one IP address? A "clean" interface would be to send only the resolved address and let the DA do the MX lookup etc. However, for piggybacking the QMGR needs to know the open connections and it must be able to compare those connections with the entries in the EDB. Hence the QMGR must do the MX expansion (using the AR or DNS helper processes).

Very simple outline of selection of next hop(s) for SMTP delivery:

1. Determine DA, destination host, address (done by AR). Let's assume the DA is a SMTP client (the first sendmail X prototype will only support SMTP and LMTP).
2. Lookup MX records for destination host. If none: only destination host (priority 0) in list.
3. Is any local host in list? If yes: remove all entries with same priority of higher.
4. Is the list of MX records empty? If yes: failure (maybe configuration option).
5. Randomize equal priority MX records; try to use an existing randomization[3] if it matches an existing set of MX records, otherwise we need functions on sets, at least for comparison. For example, if one lookup returns A and B and another one returns B and A, then a list comparison will fail, but comparing the two sets will return that they are equal. It would also be helpful to have a "is subset of" function (if the lists are sorted according to the same criteria, then sets a "is prefix of" function).
6. Lookup address records for each host in the list. Question: who does this? Will the AR return the entire list or will the QMGR cache that data and access it directly (asking a DNS helper if entries are expired)?

The MX list and the address list are represented as linked lists, with two different kind of links: same priority, lower priority. This can be either coded as two pointers or as a "type of link" value. If we use two pointers then we have to decide whether we fill in both pointers (if a record of that type is available) or only one. For example, let A, B, and C be MX records for a host with value 10, 10, and 20 respectively. Does A have only a (same priority) pointer to B, or does it have pointers to B and C? Is there a case where we do not go sequentially through the list? Maintaining two pointers is more effort which may not give us any advantage at all.

3.4.5.1 QMGR - Delivery Agents API

The QMGR provides APIs for the different modules with which it communicates. The two most important interfaces are those to the SMTP server and the delivery agents, the former is discussed in Section 3.4.12.

[3]sendmail 8 uses mxrand() to compute a hash of the host name which then is used to sort the records (as second order key after priority). This assures that the same order is used every time, hence it may not really fulfill the RFC requirement.

Question: how does the QMGR control the DAs? That is, who starts a new DA if necessary? This should be the MCP, since it starts (almost) all sendmail X processes. However, what if a DA is already running and the QMGR just wants more? Does the MCP start more or does a DA fork()? Probably the former, which means the QMGR and the MCP must be able to communicate this type of data. Do the DAs terminate themselves after some (configurable) idle time? That should be a configuration option in the MCP control file, see Section 3.3.4.

Question: does the QMGR have each of the following functions per DA or are these generic functions which take the name/type of the DA as argument and are dispatched accordingly?

General side note about (communication) protocols: it seems simpler that the caller generates the appropriate handles instead of the callee. The caller has to pass some kind of token (handle/identifier) anyway to the callee to associate the result it is getting back with the correct data (supplied via the call). If this would be a synchronous call, then the callee could generate the handle, but since we must deal with asynchronous calls, we must either generate the handle ourselves such that the callee can use it to identify the returned data, or the calling mechanism itself must generate the data, which however makes this too complicated.

- qmgr_da_new(IN da, IN da-descripton, OUT da-handle, OUT status)

  Create/request a new DA (or maybe several). The properties etc are defined in da-descripton.

- qmgr_da_new_result(IN da, IN da-handle, OUT status)

  Get result(s) from creating/requesting a new DA.

- qmgr_da_stop(IN da, IN da-handle, OUT status)

  Stop a DA (all DAs of that type).

- qmgr_da_stop_result(IN da, IN da-handle, OUT status)

  Get result of stopping a DA.

- qmgr_da_sess_new(IN da, IN session, IN da-session-handle, IN transaction, IN da-trans-handle, OUT session-status, OUT trans-status, OUT status)

  Open a connection for delivery and send one mail. session contains the destination host to which to connect and some requirements, e.g., STARTTLS, AUTH. If we only want to open a connection, transaction and da-trans-handle can be NULL.

- qmgr_da_sess_new_result(IN da, IN da-session-handle, IN da-trans-handle, OUT session-status, OUT trans-status, OUT status)

  Get results of opening a connection for delivery and sending one mail. The status can be fairly complicated since the operation can fail for various reasons in different stages, see 3.8.4.1.

- qmgr_da_ta_new(IN da, IN da-session-handle, IN transaction, IN da-trans-handle, OUT trans-status, OUT status)

  Perform one delivery (maybe to multiple recipients), transaction contains necessary information for session.

- qmgr_da_ta_new_result(IN da, IN da-session-handle, IN da-trans-handle, OUT trans-status, OUT status)

  Get results for delivery attempt. This can be a state for the entire transaction, or per recipient depending on the DA and the actual delivery.

  Notice: da-session-handle might not be necessary, da-trans-handle is sufficient to identify the transaction. However, an implementation might prefer to get also the session handle.

- qmgr_da_session_close(IN da, IN da-session-handle, OUT status)

  Close a connection (session).

- qmgr_da_session_close_result(IN da, IN da-session-handle, OUT status)

  Get result of closing a connection (session).

Notice: the handles (identifiers) from the DA (da-trans-handle, da-session-handle) are not related to the transaction/session identifiers of SMTPS. That is, we do not "reuse" those identifiers except for transaction identifiers for the purpose of logging. We only need those handles to identify the sessions/transactions in SMTPC and QMGR, i.e., to associate the data structures (and maybe threads) that describe the sessions/transactions. We can generate the identifiers in a DA (SMTPC) similarly as in SMTPS; the differences are:

- the sessions/transactions come from QMGR instead of an (external) client over SMTP,
- the identifiers are ephemeral, they don't need to be stored beyond their lifetime. Hence in theory we could reuse them, e.g., just use a thread identifier, however, unique identifiers over the life of a process have the advantage of minimizing possible confusion due to reuse if that isn't properly synchronized between QMGR and DA.

See also Section 3.1.1.2.

3.4.6 Transferring Data between EDBs

There are several different EDBs in the QMGR: active queue (AQ or ACTEDB), incoming queue (INCEDB; two variants: IBDB: backup on disk and IQDB: in memory only), and main (deferred) queue (DEFEDB). Data must be transferred between those DBs in various situations, e.g., for scheduling data is taken from IQDB or DEFEDB and put into ACTEDB. Doing so involves copying of data and maybe allocating memory for referenced data structures, e.g., mail addresses, and then copying the data from one place (`Src`) into another (`Dst`). This problem can be solved in two ways:

1. moving the pointer to the data from the old structure `Src` to the new one `Dst`, i.e., copy the pointer and set it to `NULL` in `Src`. This only works if the data isn't needed anymore in `Src`.
2. Use data structures with reference counts, see Section 3.16.7.1. This is a clean solution but adds overhead to data structures that barely need the feature (and may require additional locking).

Another approach is to use the same data structures for all/most EDBs with an additional type field that defines which data elements are valid. This way copying is either not necessary or can be done almost one-to-one in most cases. The disadvantage of this approach is the potential waste of some memory and fewer chances for typechecks by the compiler (however, more generic functions might be possible). Moreover, the data requirements for incoming and outgoing envelopes are fairly different, so maybe those should be separate data structures.

Maybe only ACTEDB and DEFEDB data structures should be identical, or at least ACTEDB structs should be a superset of DEFEDB structs. Otherwise we need to update entries in DEFEDB by reading the data in (into a structure for DEFEDB), modifying the data (according to the data in ACTEDB), and writing the data out.

3.4.6.1 Detailled Data Flow for Cut-Through Delivery

Section 2.4.3.3 describes the data flow of envelopes between the various EDB that QMGR maintains, while Section 2.4.3.6 describes the changes required for cut-through delivery. This section specifies the functional behavior for the latter.

If the final data dot is received by the SMTP server, it sends a CTAID record to QMGR including the information that this transaction is scheduled for cut-through delivery. Such an information is given by QMGR in reply to RCPT records: if all of them are flagged by QMGR for cut-through delivery, the transaction is scheduled for it. After receiving CTAID QMGR decides whether the scheduler will go ahead with cut-through delivery. If it doesn't, it sends back a reply code for case 1b below and proceeds as for normal delivery mode. Otherwise all recipients are transferred to AQ immediately, the recipients and the transaction are marked properly and delivery attempts are made. Moreover, a timeout is set for the transaction after which case 1b is selected.

Question: should the data be in some IBDB list too?

1. QMGR can return one of the following replies:
   (a) accept without `fsync(2)`: the mail has been successfully delivered to all recipients. All recipients and the transaction data are removed from IQDB and from AQ.
   (b) accept with `fsync(2)`: the mail has not been successfully delivered to all recipients. QMGR must store the information about the transaction safely in IBDB (or DEFEDB) before sending this reply.
   (c) reject
2. QMGR does not reply within the timeout: return a temporary error to the client.

For case 1b the SMTP server needs to send another message to QMGR telling it the result of `fsync(2)`. If `fsync(2)` fails, the message must be rejected with a temporary error, however, QMGR may already have delivered the mail to some recipients, hence causing double deliveries.

3.4.6.2 Reading Entries from Deferred Queue

To minimize disk I/O an envelope database cache (EDBC) is used (see Section 3.4.6.2). As explained in Section 2.4.3.5.1 the cache may not always contain all references to DEFEDB due to memory restrictions. Such a restriction can be either artificially enforced (by specifying a maximum number of entries in EDBC) or indirectly if the program runs out of memory when trying to add a new entry to the cache (see also Section 3.4.17.1). In that case, the operation mode of reading entries from the deferred queue must be changed (from *cached* to *disk*). In the disk mode the entire DEFEDB is read at regular intervals to fill up EDBC with the *youngest* entries which in turn are read at the appropriate time from DEFEDB into AQ. *Question*: how can those "read the queue" operations be minimized? It is important that EDBC is properly maintained, it contains the "youngest" entries, i.e., all entries in DEFEDB that are not referenced from EDBC have a "next time to try" *ntt* that is larger than the last entry in EDBC. *Question*: how can this be guaranteed? Proposal: when switching from *cached* to *disk* set a status flag to keep track of the mode, and store the maximum *ntt* in *nttmax*. When inserting entries into EDBC, ignore everything that has a *ntt* greater then *nttmax*. If entries are actually added ($ntt < nttmax$) and hence older entries are removed, set a new *nttmax*. Perform a DEFEDB scan when EDBC is empty or below some threshold, e.g., only filled up to ten per cent and $nttmax - now < 10$. If all entries from DEFEDB can be read, reset the mode to *cached*.

3.4.7 Reconstructing Data from IBDB

In case of an unclean shutdown there might be open transactions in IBDB. Hence on startup the QMGR must read the IBDB files and reconstruct data as necessary. The API for IBDB is specified in Section 3.13.4.3. It allows an application to read all records from the IBDB. To reconstruct the data, the function sequentially reads through IBDB and stores the data in an internal data structure (in the following called RecDB: Reconstruction DB)that allows access via transaction/recipient id. The entries are ordered, i.e., the first entry for a record has the state *open*, the next entry has the state *done* (with potentially more information, e.g., delivered, transferred to DEFEDB due to a temporary/permanent failure, or cancelled). The second entry for a record might be missing which indicates an open transaction that must be taken care of. For each done transaction the corresponding open entry is removed from RecDB. After all records have been read RecDB contains all open transactions. These must be added to DEFEDB or AQ. If they are added only to the latter, then we still need to keep the IBDB files around until the transactions are *done* in which case a record is written to IBDB. This approach causes problems if the number of open transactions exceeds the size of AQ in which case an overflow mechanism must set in, e.g., either delaying further reading of IBDB or writing the data to DEFEDB. In the first sendmail X version the data should be transferred to DEFEDB instead for simplicity. Even with this simpler approach there are still some open problems:

1. The most important is the order of operations: should the reconstruction of the open transactions be done concurrently with the normal operation of QMGR?

2. A smaller problem is that IBDB does not contain all data that is necessary for DEFEDB entries (see Sections 3.4.10.4 and 3.4.10.6).

About 1: If yes, a faster startup time is achieved since the QMGR does not need to wait for the reconstruction. This of course causes other problems, e.g., what to do if the reconstruction runs into problems[4]? There are further complications with the order of operations: the reconstruction must be performed before the new IBDB is opened unless either a different name is used or the sequence numbers start after the last previously used entry. In the former case some scheme must be used to come up with names that will not cause problems if the system goes down while the reconstruction is still ongoing. In the latter case the problem of a wrap-around must be dealt with, i.e., what happens if the sequence number reaches the maximum value? A simple approach would be to simply start over at 1 again, but then it must be avoided that those files are still in use (which seems to be fairly unlikely if for example a 32 bit unsigned integer is used because the number of files would be huge, definitely larger what is sane to store in a single directory[5]).

A potential solution to the problem of overlapping operation is to use different sets of files, e.g., two different directories: *ibdbw* for normal operation, *ibdbr* for recovery. In that case at startup the following algorithm is used: If both ibdbw and ibdbr exist then the recovery run from the last startup didn't finish properly. Hence the rest of QMGR is not started before completing recovery, i.e., asynchronous operation is only allowed if the previous recovery succeeded. This simple approach allows us to deal with recovery problems without introducing an unbound number of IBDB file sets (directories). If only ibdbw exists, then rename it to ibdbr and let the QMGR startup continue while recovery is running. After recovery finished, ibdbr is removed thus indicating successful recovery for subsequent startups.

This approach can be extended to deal with a *cleanup* process that "compresses" IBDB files by removing all closed transactions from them. This cleanup process uses a different directory *ibdbc* in which it writes open transactions similar to the recovery program. That is, it reads IBDB files from ibdbw, gets rid of

[4]This might be similar to background `fsck` in recent FreeBSD 5 versions
[5]A simple hashing scheme could be used

closed transaction in the same way as the recovery function does and writes the open transactions into a new file. After this has been safely committed to persistent storage, the IBDB files that have been read can be removed. The recovery function needs to read in this case the files from ibdbw and ibdbc to reconstruct all open transactions. The cleanup process should minimize the amount of data to read at startup. Such a cleanup function may not be available in sendmail X.0, however, if the MTA runs for a long time it may require a lot of disk space if there is not cleanup task. *Question*: is cleaning up a recursive process? If so, how to accomplish that? In a simple approach two "versions" can be used between which the cleanup task toggles, i.e., read from version 1 and write to version 2, then switch: read from version 2 and write to version 1. Question: how easy is it to keep track of this?

An example of the missing data mentioned as problem 2 in the list of problems is the start time (of the transaction) which is not recorded. This can be either taken in first approximation from the creation time of the IBDB file, or the current time can be simply used (which might be off by a fair amount if the system is down for a longer time, which, however, should not happen).

3.4.7.1 Cleaning up IBDB

Question: is there a way to avoid writing data when cleaning up IBDB? The cleanup task could "simply" remove IBDB files that contain only references to closed transactions. We may not even have to read any IBDB files at all since the data can be stored in IQDB, i.e., a reference to the IBDB logfile (sequence number). This requires a reference counter for each IBDB logfile which contains the number of open transactions (TA/RCPTs). When both reference counters reach zero the logfile can be removed. Note: this violates the modularity: now IQDB is used to store data that is related to the implementation of IBDB, i.e., both are tied together. Currently IQDB is fairly independent of the implementation of IBDB, e.g., it does not know about sequence numbers. Now there must a way to return those numbers to IBDB callers and they must be stored in IQDB. Moreover, there are functions that do not allow for a simple way to do this, e.g., those which act on request lists. In this case it would be fairly complicated to associate the IBDB data with the IQDB data. However, at the expense of memory consumption, the data could be maintained by the IBDB module itself. In this case it behaves similar as the IBDB recovery program, i.e., it stores open transactions (only a subset of data is necessary: identifier and sequence number) in an internal (hash) table, and matches closed transactions against existing entries to remove them. Periodically it can scan the table to find out which sequence numbers are still in use and remove unused files.

A different (simpler?) approach to the problem is to use time-based cleanup. Open transactions that are stored in IBDB are referenced by IQBD (at least with the current implementation) or AQ (entries which are marked to come from IQDB). Periodically these entries can be scanned to find the oldest. All older IBDB logfile can be removed. Note: there should be a correctness proof for this before it is implemented.

3.4.8 Deferred Envelope Database: Recipient Addresses

An interesting question is in which format recipient addresses are stored in the EDB, i.e., whether only the original recipient address is stored or whether also the resolved format (DA, host, address, etc) is stored too. If we use the resolved (expanded) format then we need to remove/invalidate those in case of reconfigurations. These reconfigurations may change DAs or routing. A possible solution is to add a time stamp to the resolved form. If that time stamp is older than the last reconfiguration then the AR must be called again. However, the routing decision may have been based on maps which have changed inbetween, hence this isn't a complete solution. It may require a TTL based on the minimum of all entries in maps used for the routing decision. Alternatively we can keep the resolved address "as-is" and do not care about changes. For examples, some address resolution steps happen in sendmail 8 before the address is stored in the queue, some happen afterwards. Examples for the former are alias expansion, which certainly should not be done every time. So the only address resolution that happens during delivery are DNS lookups (MX records, A/AAAA records), and those can be cached since they provide TTLs. We might make the address resolution a clean two step process:

1. address resolution which is done only once before the (resolved) recipient address is stored in the queue;
2. address resolution which is done during each delivery attempt and hence changes take effect almost immediately for all (not just new) recipients.

It might be an interesting idea to provide a cache of address mappings. However, such a cache cannot be simply for domains since it might be possible to provide per-address routing. The cache may be for DNS (MX/A/AAAA) lookups nevertheless, i.e., a "partially" resolved address maps to a domain which in turn maps to a list of destination hosts. This is fairly much what sendmail 8 does:

1. the address rewriting engine returns a triple mailer, host, and address.
2. The host part is mapped to list of destination hosts during "final" delivery via MX lookups (unless suppressed via a flag or other measures).

Note that the host part can also be a list of (colon separated) hosts.

These two steps are clearly related to the two steps listed above.

Incomplete (as of now) summary: Advantages of storing resolved addresses:

- direct access, no need to call AR again unless address is "expired" or some reconfiguration happened.

Disadvantages of storing resolved addresses:

- harder to deal with configuration changes.
- more data must be stored and it is maybe replicated several times in DEFEDB.

If an address "expires" earlier than the computed "next time to try" then it probably is not useful to store the resolved address in DEFEDB. However, if the scheduler decides to try the address before the "next time to try", e.g., because a host is available again, then the resolved address might still be useful.

See also Section 3.4.15 for further discussion of this problem.

We also need to mark addresses (similar to sendmail 8 internally) to denote their origin, i.e., original recipient, expanded from alias (alias or list), etc.

3.4.9 Scheduler Algorithms

As described in Section 2.4.7 the scheduler must control the load it generates.

3.4.9.1 Slow Start and Connection Limit

One of the algorithms the scheduler should implement is "slow start", i.e., when a connection to a new host is created, only a certain number of initial connections must be made ("initial concurrency"). When a session/transaction has been succesful, the number of allowed concurrent connections can be increased (by one for each successful connection) until an upper limit ("maximum concurrency") is reached. This varying number of concurrency is usually called "window" (see TCP/IP). If a connection fails, the size of the window is decreased until it reaches 0 in which the destination host is declared "dead" (for some amount of time).

Question: which kind of failures should actually decrease the window size besides any kind of network I/O errors? For example, a server may respond with `452 Too many connections` but smX will not try to interpret the text of the reply, only the reply code.

3.4.10 Data Structures

The queue manager uses several data structures to store the status of the system and the envelopes of mails for which it has to schedule delivery. These are listed in the next sections.

3.4.10.1 Connection Cache Access

Question: are the connection caches indexed on host name or IP address? Problem: there is no 1-1 relation between host names and IP addresses, but an N-M relation. This causes problems for finding recipients to send over an open connection. A recipient address is mapped to (a DA and) a destination host, which is mapped to a list of MX records (host names) which in turn are mapped to IP addresses. Since there can be multiple address records for a host name and multiple PTR records for an IP address, we have a problem. The general problem is what to use as index for the connection caches. A smaller problem results from the expiration (TTL) of DNS entries (all along the mapping: MX, A/AAAA, and PTR records). Question: do load balancers further complicate this or can we ignore them for our purposes? Possible solution: use host name as index, provide N-M mappings for host name to IP address and vice versa. These mappings are provided by DNS. A simpler (but more restrictive solution) is to use the IP address and the host name together as index (see Exim [Haz01]). Question: do we want our own (simpler) DNS interface? We need some (asynchronous) DNS "helper" module anyway, maybe that can add some caching and a useful API? We shouldn't replicate caches too often due to the programming and memory usage overhead. So how do we want to access the connection caches? If a host name maps to several IP addresses, must it be a single machine? Does it matter wrt SMTP? Is a host down or an IP address? It could be possible that some of the network interfaces are down, but the host still can receive e-mail via another one.

So the QMGR to DA interface should provide an option to tell the DA to use a certain IP address for a delivery attempt because the QMGR knows the DA has an open connection to it. Even though this is a slight violation of the abstraction principle, the QMGR scheduled this delivery because of the connection cache, so it seems better than letting the DA figuring out to use one of its open connection (by looking up addresses etc).

3.4.10.2 Connection Reuse Problem

Problem: a connection (session) has certain attributes, e.g., STARTTLS, AUTH restrictions/requirements. These session-specific options can depend on the host name or the IP address (in sendmail 8: TLS_Srv:, AuthInfo:). This makes it even more complicated to reuse a connection. If a host behaves differently based on under which IP address it has been contacted, or if different requirements are specified for host name/IP addresses then connection reuse is significantly more complicated. In the worst case this could result in bounces that happen only in certain situations. It is certainly possible to document this behavior on the sendmail side, but if the other (server) side has a connection information dependent behavior then we have a problem.

A connection is made to an IP address, the server only knows the client IP address (plus port, plus maybe ident information, the latter should not be used for anything important). SMTP doesn't include a "Hi, I would like to speak with X" greeting, but only a "Hi, this is Y" (which might be considered a design flaw), so the server can't change its behavior based on whom the client wants to speak to (which would be useful for virtual hosting), but only based on the connection information (IP addresses, ports). Hence when a connection is reused (same IP address) the server can't change its behavior. Problem solved? Not really, someone could impose wierd restrictions based on sender addresses. It seems to be necessary to make this configurable (some external map/function can make the decision). This probably can be achieved by imposing a transactions per session limit, see also Section 2.4.8.

Another solution might be to base connection reuse on host name and IP address[6]. This may restrict connection reuse more than necessary, but it should avoid the potential problems. Maybe that should be a compile time option? Make sure that the code does not depend completely on the type of the index.

There have been requests to perform SMTP AUTH based on the sender address. which of course invalidates connection reuse. It's one (almost valid) example for additional requirements for a connection. It's only almost valid, since SMTP AUTH between MTAs is not for user authentication, it is used to authenticate the directly communicating parties. However, SMTP AUTH allows some kind of proxying (authenticate as X, authorize to act as Y), which seems to be barely used.

Note: connection reuse requires that the delivery agent is the same, if two recipients resolve to different delivery agents — even for the same IP address and hostname — then the connection will not be reused. In some cases this seems rather useless[7] hence the maximum flexibility would be reached by establishing congruence classes of mailer with respect to connection reuse. If then not just the mailer definitions are used to create those classes but also some connection attributes (see begin of this section), then we may have found the right approach to connection reuse.

3.4.10.3 Data for DSN

See Section 2.4.6 for the different types of DSNs and the problems we may encounter. We need to store (at least?) three different counters as explained in Section 2.4.6.1. DSNs can be requested individually for each recipient. Hence the sum of these counters is less than or equal the number of original recipients. Question: could it ever increase due to alias expansion? We could store the number of requested DSNs for each type and then compare the number of generated DSNs against them. If any of the generated DSN counters reached the requested counter we can schedule the DSN for delivery and hence we can be sure (modulo the question above) that all DSNs of the same type can be merged into one. Question: do we really need those counters? Or do we just generate a DSN whenever necessary and schedule it for delivery (with some delay)? Then the DSN generation code could look whether there is already a DSN generated and add the new one to it, as far as this is possible since the scheduler has to coalesce those DSNs. Problem: there is extra text (the error description) that should be put into one mail. How to do this? See also Section 2.4.6 about the question how to generate the body of DSNs.

3.4.10.4 Incoming Queue (INCEDB)

As explained in Section 2.4.1, the incoming queue consists of two parts: a restricted size cache in memory and a backup on disk.

[6] This is what exim does.

[7] See sm8: there's not much difference between the mailers `relay` and `esmtp`.

The incoming queue does not (need to) have the same amount of data as the SMTP servers. It only stores data that is relevant to the QMGR. There is even less information in the data that is written to disk when an entry has been received. The data in the RSC is not just needed for delivery, but also during mail reception for policy decisions. In contrast, the backup data is only there to reconstruct the incoming cache in case of a system crash, i.e., the mail has already been received, there will not be any policy decisions about it. Hence the backup stores only the data that is required for delivery, not the data that is necessary to make decision while the mail is being received. This of course means that a reconstructed RSC does not contain the same (amount of) information as the original RSC.

The size of the cache defines the maximum number of concurrently open sessions and transactions in the SMTP servers. Question: do we store transactions and recipients also in fixed size caches? We certainly have to limit the size used for that data, which then acts as an additional restriction on the number of connections, transactions, and recipients. It's probably better to handle at least the recipients dynamically instead of pre-allocating a fixed amount of memory. The amount of memory must be limited, but it should be able to shrink if it has been expanded during a high volume phase (instead of having some maximum reserved all the time). Since most of the information in the cache is not fixed size, we need to dynamically allocate memory anyway. We maybe need some special kind of memory allocation for this which works within a given allocated area (see also Section 3.16.6).

Each entry in the cache has one of the following two formats (maybe use two different RSC):

Session

session-id	session identifier
client-host	identification of connecting host
	IP address, host name, ident
features	features offered: AUTH, TLS, EXPN, ...
workarounds	work around bugs in client (?)
transaction-id	current transaction
reject-msg	message to use for rejections (needed?)
auth	AUTH information
starttls	TLS information
n-bad-cmds	number of bad SMTP commands
n-transactions	number of transactions
n-rcpts	total number of recipients
n-bad-rcpts	number of bad recipients

Transaction:

transaction-id	transaction identifier
start-time	date/time of transaction
mail	address, arguments (decoded?)
n-rcpts	number of recipients
rcpt-list	addresses, arguments (decoded?)
cdb-id	CDB identifier (obtained from cdb?)
msg-size	message size
n-bad-cmds	number of bad SMTP commands (necessary?)
n-rcpts	number of valid recipients
n-bad-rcpts	number of bad recipients
session-id	(pointer back to) session
statistics:	
end-time	end of transaction

If recipients addresses are expanded while in the INCEDB, we need to store the number of original recipients too.

Backup on disk: those entries have a different format than the in-memory version. The entries must be clearly marked as to what they are: transaction sender or recipient.

Sender (transaction):

transaction-id	transaction identifier
start-time	start time of transaction
sender-spec	address incl. ESMTP extensions
cdb-id	CDB identifier (obtained from cdb?)
n-rcpts	reference count for cdb-id

Notice: the sender information is written to disk after all recipients have been received, i.e., when DATA is received, because it contains a counter of the recipients (reference count).

ESMTP sender extensions (substructure of the structure above)

size	size of mail content (SIZE=)
bodytype	type of body
envid	envelope id
ret	DSN return information (FULL, HDRS)
auth	AUTH parameter
by	Deliverby specification

Per recipient (transaction):

transaction-id	transaction identifier
rcpt-spec	address incl. ESMTP extensions
	and maybe a unique id (per session/transaction?)

ESMTP Recipient extensions (substructure of the structure above):

notify	DSN parameters (SUCCESS, FAILURE, WARNING)
orcpt	original recipient

The data for sender and recipient should be put into a data structure such that all relevant data is kept together. That structure must contain the data that must be kept in (almost) all queues.

3.4.10.5 Active Queue (ACTEDB, AQ)

The active queue needs different types of entries, hence it might be implemented as two RSCs, i.e., one for senders and one for recipients[8]. It could also be only one RSC if the "typed" variant is used (see 4.3.5.1).

Notice: there are two types of transaction records:

1. The (incoming, SMTPS) transaction.

2. The (outgoing, DA) transaction.

Since sendmail X performs scheduling per recipient (see Section 2.1.5, item 5), not per (incoming) transaction, those two transactions do not need to match. The outgoing (DA) transaction context is not stored

[8] Currently it is implemented as two linked lists, one each for transactions and recipients.

directly in AQ, it is part of the QMGR - DA interface, see 3.4.10.12. However, the recipient data in AQ has data that refers to an outgoing transaction as described below.

The AQ context itself contains some summary data and the data structures necessary to access transactions and recipients:

max-entries	maximum number of entries
limit	current limit on number of entries
entries	current number of entries
t-da	entries being delivered
nextrun	if set: don't run scheduler before this time
tas	access to transactions
rcpts	access to recipients

There should probably be more specific counters: total number of recipients, number of recipients being delivered, number of recipients waiting for AR, number of recipients ready to be scheduled, and total number of transactions.

The incoming transaction context contains at least the following fields:

transaction-id	SMTPS transaction identifier
sender-spec	address (see INCEDB)
cdb-id	CDB identifier
from-queue	from which queue does this entry come? (deferred or incoming)
counters	several counters to keep track of delivery status

For ACTEDB we only need to know how many recipients are referenced by this transaction in the DB itself. That is, when we put a new recipient into ACTEDB, then we need to have the sender (transaction) context in it. If it is not yet in the queue, then we need to get it (from INCEDB or ACTEDB) and initialize its counter to one. For each added recipient the counter is incremented by one, when the status of a recipient delivery attempt is returned from a DA, the counter is decremented by one and the recipient is taken care of in the appropriate way. See also 2.4.3.4. However, because AQ should contain all necessary data to update DEFEDB it must also store the overall counters, e.g., how many recipients are in the system in total (not just in AQ).

Note: the delivery status for a transaction is not stored in DEFEDB since each delivery attempt in theory may lead to a different transaction, i.e., a DA transaction is not stored in DEFEDB.

The recipient context in AQ contains at least the following elements:

SMTPS transaction-id	SMTPS transaction identifier
DA transaction-id	DA transaction identifier
rcpt-spec	address (see INCEDB)
rcpt-internal	delivery tuple (DA, host, address)
from-queue	from which queue does this entry come?
status	not yet scheduled, being delivered, (temp) failure, ...
SMTPS-TA	recipient from same SMTPS transaction
DA-TA	recipient in same DA transaction
DEST	recipient for same destination

The last three entries are links to access related recipients. These are used to group recipients based on the usual criteria, i.e., same SMTPS transaction, same delivery transaction, same next hop. Maybe this data can also be stored in DEFEDB to pull in a set of recipients that belongs together instead of searching during each scheduling attempt for the right recipients that can be grouped into a single transaction. *Questions*: how to do this? Is it worth it?

The internal recipient format is the resolved address returned by the AR. Its format is explained in Section 3.6.3.1.

As explained in 3.4.4, several access methods are required for the various EDBs, those for AQ are:

1. The unique identifier for entries in AQ is the SMTPS transaction identifier (plus recipient index for recipients).

2. Another (non-unique) key is the DA transaction identifier, which is required to associate DA results with entries in AQ; as explained in Section 3.4.10.12 the DA status cache stores also DA and SMTPS transaction identifier.

3. The DA and next hop are used as key (maybe combined into one) to access a list of recipient entries which are going to be delivered to the same destination (see Section 2.4.4.8: AQRD, 3). This data access method can also be used to easily control the number of allowed concurrent connections (see also 3.4.9.1): when that number is reached, no entry in the list is scheduled anymore, i.e., the entire list is skipped, which is simpler than checking for each recipient whether the limit has been reached.

The key described in item 3 refers to another data structure which summarizes the entries. This data structure is the "head" of the DEST list:

DA	Delivery Agent
next-hop	Host (destination/next hop) to which to connect for delivery
todo-entries	Number of entries in todo-list
todo-list	Link to recipients which still need to be sent
busy-entries	Number of entries in busy-list
busy-list	Link to recipients which are being sent

The number of waiting transactions (todo-entries) can be used to determine whether to keep a session open or close it.

Question: is it really useful to have a busy list? What's the purpose of that list, which algorithms in the scheduler need this access method? The number of entries in the busy list is somehow useful if it were the number of open transactions or sessions, however, this is the number of recipients which does not have a useful relation to transactions/sessions.

Note: when a recipient is added to AQ it may not be in these destination queues because its next hop has not yet been determined, i.e., the address resolver needs to be called first. Those entries must be accessible via other means, e.g., their unique (recipient) identifier (see item 1 above). It might also be possible (for consistency) to have another queue with a bogus destination (e.g., a reserved DA value or IP address) which contains the entries whose destination addresses have not yet been resolved. Section 3.4.4 explains some of the problems with chosing indices to access AQ (and other EDBs), there is an additional problem for AQRD: if the connection limit for an IP address is reached, the scheduler will skip recipients for that destination. However, the recipient may have other destinations with the same priority whose limit is not yet reached. Either the system relies on the randomization of those same priority destinations (real randomization in turn causes problems for session reuse), or some better access methods need to be used. It might be useful in certain cases to look through the recipient destinations nevertheless (which defeats the advantage of this organization to easily skip entries that cannot be scheduled due to connection limits).

There might be yet another data structure which provides a summary of the entries in DEFEDB of this kind. That data structure can be used to decide whether to pull in recipients from DEFEDB to deliver them over an open connection.

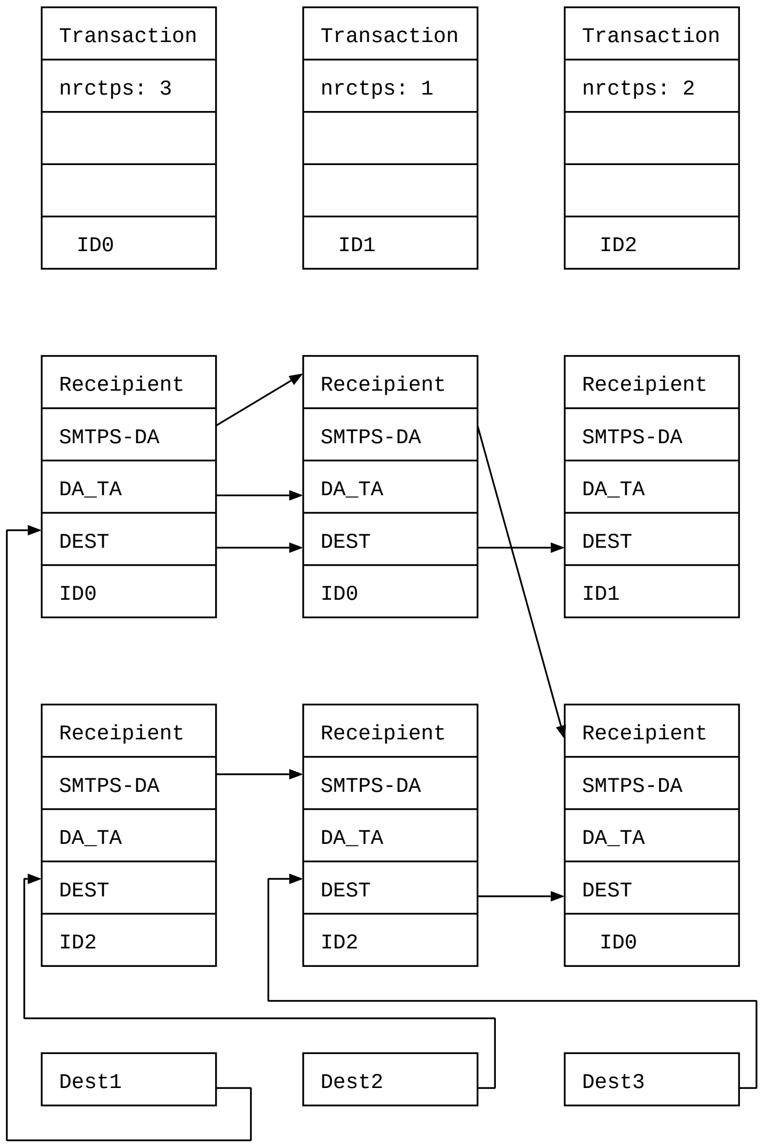

Figure 3.1: Example of Active Queue

3.4.10.5.1 Cleaning up AQ The QMGR/scheduler must also remove entries from AQ that are too long in the queue, either because AR didn't respond or because a delivery attempt failed and the DA didn't tell QMGR about it (see Section 2.4.4.5). *Question*: what is an efficient way to do this? Should those entries also be in some list, organized in the order of timeout? Then the scheduler (or some other module) just needs to check the head of the list (and only needs to wake up if that timeout is actually reached). When an item is sent to AR or a DA then it must be sorted into that list.

3.4.10.6 Deferred Queue (DEFEDB)

The entries must be clearly marked as to what they are: transaction sender or recipient.

Sender:

transaction-id	transaction identifier
start-time	date/time mail was received
sender-spec	address (see INCEDB)
cdb-id	CDB identifier
rcpts-left	reference count for cdb-id
rcpts-tried	counter for "tried" recipients

rcpts-left refers to the number of recipients which somehow still require a delivery, whether to the recipient address or a DSN back to the sender. rcpts-tried is used to determine when to send a DSN (if requested). It might be useful to have counters for the three different delivery results: ok, temporary/permanent failure:

rcpts-total	total number of recipients
rcpts-left	number of recipients left
rcpts-temp-fail	number of recipients which temporary failed
rcpts-perm-fail	number of recipients which permanently failed
rcpts-succ	recipients which have been delivered

In case of a DELAY DSN request we may need yet another counter. See also Sections 2.4.6 and 3.4.10.3.

rcpts-total is at least useful for statistics (logging); one of rcpts-succ and rcpts-total may be omitted. The invariances are:

$rcpts\_total = rcpts\_temp\_fail + rcpts\_perm\_fail + rcpts\_succ$

rcpts-total also counts the bounces that have been generated. It is never decremented.

$rcpts\_left = rcpts\_temp\_fail + rcpts\_perm\_fail$

Notice: these counters must be only changed if the delivery status of a recipient changes. For example, if a recipient was previously undelivered and now a delivery caused a temporary failure, then rcpts-temp-fail is increased. However, if a recipient previously caused a temporary failure and now a delivery failed again temporarily, then rcpts-temp-fail is unchanged. This obviously requires to keep the last delivery status for each recipients (see below).

Recipient:

transaction-id	transaction identifier
rcpt-spec	address (see INCEDB)
rcpt-internal	delivery tuple (DA, host, address, timestamp)
d-stat	delivery status (why is rcpt in deferred queue)
schedule	data relevant for delivery scheduling, e.g., last-try: last time delivery has been attempted next-try: time for next delivery attempt

d-stat must contain sufficient data for a DSN, i.e.:

act-rcpt	actual recipient (?)
orig-rcpt	original recipient (stored in rcpt-spec, see above)
final-rcpt	final recipient (from RCPT command)
DSN-status	extended delivery status code
remote-mta	remote MTA
diagnostic-code	actual SMTP code from other side (complete reply)
last-attempt	data/time of last attempt
will-retry	for temporary failure: estimated final delivery time

The internal recipient format is the resolved address returned by the AR. Its format is explained in Section 3.6.3.1. Question: do we really want to store rcpt-internal in DEFEDB? See Section 3.4.8 for a discussion. The timestamp for the delivery tuple is necessary as explained in the same section.

Question: which kind of delivery timestamp is better: last time a delivery has been attempted or time for next delivery attempt? We probably need both (last-try for DSN, next-try for scheduling).

3.4.10.7 Deferred Queue Cache (EDBC)

EDBC implements a sorted list based on the "next time to try". with references to recipient identifiers (which are the main indices to access DEFEDB). "Next time to try" is not a unique identifier hence this structure must be aware of that, e.g., when adding or removing entries.

3.4.10.8 Connection Cache (incoming)

This cache is accessed via IP addresses and maybe hostnames. It is used to check whether an incoming connection (to SMTPS) is allowed (see also Section 2.4.7).

open-conn	number of currently open connections
open-conn-X	number of open connections over last X seconds (probably for X in 60, 120, 180, ...)
trans-X	number of performed transactions over last X seconds
rcpts-X	number of recipients over last X seconds
fail-X	number of SMTP failures over last X seconds
last-conn	time of last connection
last-state	status of last connection, see 3.4.10.9

Notice: statistics must be kept in even intervals, otherwise there is no way to cycle them as time goes on.

Question: do we use a RSC for this? If so, how do we handle the case when the RSC is full? Just throwing out the least recently used connection information doesn't seem appropriate.

3.4.10.9 Connection Status (incoming)

Question: what kind of status do we want to store here? Whether the connection was succesfull, or aborted by the sender? Or whether it acted strange, e.g., caused SMTP violations? Maybe performance related data? For example, number of recipients, number of transactions, throughput, and latency.

3.4.10.10 Connection Cache (outgoing) Connections

As explained in Section 2.4.4.8, there are two different connection caches for outgoing connections: one for currently open connections (OCC, 1) and one for previously made (tried) connections (DCC, 2). These two connection caches are described in the following two subsections.

Note: it might be possible to merge these two caches into one if the proper implementation, such as a restricted size cache (see Section 3.13.10), is chosen.

3.4.10.10.1 Connection Cache for currently open (outgoing) Connections This is OCC (see Section 2.4.4.8: 1) for the currently open (outgoing) connections.

OCC helps the scheduler to perform its operation, it contains summary information (and hence could be gathered from the AQ/DA data by going through the appropriate entries[9]).

open-conn	number of currently open connections
open-conn-X	number of open connections over last X seconds (probably for X in 60, 120, 180, ...)
trans-X	number of performed transactions over last X seconds
rcpts-X	number of recipients over last X seconds
fail-X	number of failures over last X seconds
performance	data related to performance, see 3.13.8
first-conn	time of first connection
last-conn	time of last connection
last-update	time of last status update
last-state	status of last connection, see 3.4.10.11
initial-conc	initial concurrency value
max-conc	maximum concurrency limit
cur-conc	current concurrency limit ("window")

This connection cache stores only information about current connections. The connection cache also stores the time of the last connection. Question: do we need to store a list of those times, e.g., the last three? We can use these times to decide when a new connection attempt should be made (if the last connection failed). For this we need at least the last connection time and the time interval to the previous attempt. If we use exponential backup we need only those two values. For more sophisticated methods (which?) we need probably more time stamps.

It is not yet clear whether the open connection cache actually needs the values listed above, especially those for "over last X seconds". Unless the scheduler actually needs them, they can be omitted (they might be useful for logging or statistics). Instead, the counters may be for the entire "lifetime" of the connection cache entry; those counters can be used to implement limits for the total number of sessions, transactions, recipients, etc.

The last three values are used to implement the slow-start algorithm, see 3.4.9.1. The time of the last status update can be used to expire the "host is dead" marking, i.e., cur-conc equal zero.

Question: which times do we actually need: the time of the last connection or last status update? If a connection is successful, but the session takes very long, those times may differ substantially. The time of the last status update is used to expire a "host is dead" marking (as explained in the previous paragraph). The time of the last connection might be used to expire an entry if something went wrong with the delivery agent, e.g., it "died" in some way such that QMGR does not notice it and hence cannot clean up properly. In this case the expiration timeout should depend on the message size, obviously it takes much longer to deliver a message of several MB than of a few KB. The timeout could be set as the some of a basic timeout (60s?) and the message size divided by the transfer rate, where transfer rate is an option set in a configuration file (and maybe checked against the "performance" values for this entry).

3.4.10.10.2 Connection Cache for previously made Connections This is DCC (see Section 2.4.4.8: 2) for previously open connections. It contains similar data as OCC but only for connections which are not open anymore.

[9]This is the common tradeoff between storage and computation.

open-conn-X	number of open connections over last X seconds (probably for X in 60, 120, 180, ...)
trans-X	number of performed transactions over last X seconds
rcpts-X	number of recipients over last X seconds
fail-X	number of failures over last X seconds
performance	data related to performance, see 3.13.8
last-conn	time of last connection
last-update	time of last status update
last-state	status of last connection, see 3.4.10.11

The connection cache also stores the time of the last connection. Question: do we need to store a list of those times, e.g., the last three? We can use these times to decide when a new connection attempt should be made (if the last connection failed). For this we need at least the last connection time and the time interval to the previous attempt. If we use exponential backup we need only those two values. For more sophisticated methods (which?) we need probably more time stamps.

This connection cache can be optimized to ignore some recent connections at the expense of being limited in size. For example, see the BSD `inetd(8)` ([ine]) implementation which uses a fixed-size hash array to store recent connections. If there are too many connections, some entries are simply overwritten (least recent entry will be replaced).

3.4.10.11 Connection Status (outgoing)

See Section 3.8.4.1 for a delivery status that must be stored in the appropriate entries. Question: where do we store the status? We store it on a per-recipient basis in the EDB and on a per-host (or whatever the index will be) basis in the connection cache. The delivery status will be only stored in the connection cache if it pertains to the connection. For example, "Recipient unknown" is not stored in that cache. The delivery status should reflect this distinction easily. Question: is it useful to create groups of recipients, i.e., group those recipients within an envelope that are sent to the same host via the same DA? This might be useful to schedule delivery, but should we create extra data types/entries for this?

It must also be stored whether currently a connection attempt is made. This could be denoted as one open connection and status equal "Opening" (or something similar).

3.4.10.12 DA Status Cache

Question: how much should the QMGR control (know about) the status of the various DAs? Should it know exactly how many are active, how many are free? That seems to be useful for scheduling, e.g., it doesn't make sense to send a delivery task to a DA which is completely busy and unable to make the delivery attempt "now", i.e., before another one is finished. Hence we need another data structure that keeps track of each available DA (each "thread" in it, however, this should be abstracted out; all the QMGR needs to know is how many DAs are available and what they are doing, i.e., whether they are busy, idle, or free[10]). The data might be hierarchically organized, e.g., if one DA of a certain type can offer multiple incarnations, then the features of the DA should be listed in one structure and the current status of the "threads" in a field (or list or something otherwise appropriate). Some statistics need to be stored too which can be used to implement certain restrictions, e.g., limit the number of transactions/recipients per session, or the time a connection is open.

[10]free: no session active; idle: session open, but no transaction; busy: session open, transaction active.

status	busy/free (other?)
DA session-id	DA session identifier
DA transaction-id	DA transaction identifier
SMTPS transaction-id	SMTPS transaction identifier
server-host	identification of server: IP address, host name
n-trans	number of performed transactions
n-rcpts	number of recipients
n-fail	number of failures
performance	data related to performance, see 3.13.8
opened	time of connection open
last-conn	time of last connection

Question: what do we use as index to access this structure? We could use a simple integer DA-idx (0 to max threads-1), i.e., a fixed size array. Then however we should also use that value as an identifier for communication between QMGR and DA, otherwise we still have to search for session/transaction id. Nevertheless, using an integer might give us the wrong idea about the level of control of the QMGR over the DA, i.e., we shouldn't assume that DA-idx is actually useful as an index in the DA itself.

Notice: this is the only place where we store information about a DA session, the active queue contains only mail and recipient data. Hence we may have to store more information here. This data structure is also used for communication between QMGR and DAs; it associates results coming back from DAs (which use DA session/transaction ids) with the data in AQ (which use SMTPS session/transaction ids).

3.4.10.13 Load Control Data Structures

As explained in Section 2.4.7 the QMGR must store information about the local load of the system. Question: how to represent this?

One simple approach is to check how much storage resources are used, e.g., how full are AQ, IQDB, etc, as well as disk space usage. However, that does not take into account the "load" of the system, i.e., CPU, I/O, etc.

3.4.10.13.1 Disk Space Usage Various DBs are stored on disk: CDB, DEFEDB, and IBDB. The latter two are completely under control of QMGR, the former is used by SMTPS (write), DA (read), and QMGR (unlink). The amount of available disk can be stored in a data structure and updated on each operation that influences it. Additionally system calls can be made periodically to reflect changes to the disk space by other processes. About CDB: SMTPS should pass the size of a CDB entry to QMGR which then can be used when a transaction is accepted and when all recipients for a transaction have been delivered and hence the CDB entry is removed.

sendmail 8.12 uses a data structure to associate queue directories with disk space ("partitions"). A similar structure can be used for smX.

```
struct filesys_S {
   dev_t           fs_dev;        /* unique device id */
   long            fs_kbfree;     /* KB free */
   long            fs_blksize;    /* block size, in bytes */
   time_T          fs_lastupdate; /* last time fs_kbfree was updated */
   const char     *fs_path;       /* some path in the FS */
};
```

3.4.10.13.2 Cache Usage For internal (memory resident) DBs it is straightforward to use the number of entries in the DB as a measure for its usage. This number should be expressed as percentage to be independent of the actual size chosen at runtime. Hence the actual usage of a DB can be represented as a single number whose value ranges from 0 to 100.

3.4.11 Database and Cache APIs

See Section 3.13.3 for more information about envelope databases, esp. APIs.

3.4.12 QMGR - SMTPS API

Notice: these functions are "called" from SMTPS (usually via a message), hence they do not have a corresponding function that returns the result. The functions may internally wait for the results of others, but they will return the result "directly" to the caller, i.e., via a notification mechanism. The other side (SMTPS) may use an asynchronous API (invoke function, ask for result) as explained in Section 3.1.1.

- qmgr_smtps_open(IN smtps-info, OUT status): a new SMTPS is started, give information about this server to the QMGR.
- qmgr_session_open(IN connection-info, IN session-id, OUT status): create a new SMTP session in the incoming cache (API see Section 3.13.4). This may reject a connection based on policy (e.g., too many connections open, load too high).
- qmgr_session_status(IN connection-info, IN session-id, IN session-status, OUT status): update session status, e.g., STARTTLS, AUTH.
- qmgr_trans_open(IN session-id, IN sender-env-info, IN trans-id, OUT status): create a new envelope (done on MAIL command).
- qmgr_rcpt_add(IN trans-id, IN rcpt-env-info, OUT status): add a new recipient (RCPT command). Maybe this should also be allowed to add a list of recipients?
- qmgr_cdb(IN trans-id, IN cdb-id, OUT status): add cdb identifier (BODY command). This could be merged with the transaction close, unless we want to allow for interleaved delivery.
- qmgr_trans_close(IN trans-id, IN cdb-id, OUT status): close a transaction (cdb-id is optional if already supplied earlier on). At this moment the QMGR can schedule the recipients of this envelope for delivery.
- qmgr_trans_discard(IN trans-id, OUT status): discard envelope (connection has been aborted somehow: RSET, EHLO, connection error on lower level).
- qmgr_session_close(IN session-id, OUT status): close a session.
- qmgr_smtps_close(IN smtps-info, OUT status): a SMTPS shuts down.

Maybe qmgr_trans_close() and qmgr_trans_discard() can be merged into one function which receives another parameter: qmgr_trans_close(IN trans-id, IN cdb-id, IN smtps-status, OUT status) that determines whether SMTPS has accepted so mail so far; the QMGR can still return an error.

3.4.13 QMGR - First Level Scheduler API

See Section 2.4.4.2 for a description of the tasks of the first level scheduler. This part of the QMGR adds entries to the active queue, whose API is described in Section 3.13.5.

qmgr_fl_getentries(IN actq, IN incq, IN defq, IN number, IN policy, OUT status) get up to a certain number of entries for the active queue.

qmgr_fl_get_inc(IN actq, IN incq, IN number, IN policy, OUT status) get up to a certain number of entries for the active queue from the incoming queue.

qmgr_fl_get_def(IN actq, IN defq, IN number, IN policy, OUT status) get up to a certain number of entries for the active queue from the deferred queue.

qmgr_fl_get_match(IN actq, IN number, IN criteria, OUT status) get some entries for the active queue from the deferred queue that match some criteria, e.g., items on hold with a matching hold message, or items for a certain domain. We may want different functions here depending on what a user can request. But we also want a generic function that can get entries depending on some conditions that can fairly freely specified.

3.4.14 QMGR - Delivery Scheduler API

See Section 2.4.4.3 for a description of the tasks of the second level (micro) scheduler. This part of the QMGR controls the active queue, whose API is described in Section 3.13.5. It uses the first level scheduler to fill the active queue whenever necessary, and the DA API (3.4.5.1) to request actual deliveries.

Question: which additional functions do we need/want here?

Whenever a delivery attempt has been made, the status will be collected and an appropriate update in the main (or incoming) queue must be made.

Common to all cases is the handling of DSNs. If a particular DSN is requested and the conditions for that DSN are fulfilled, then the recipient is added to the DSN for that message (based on the transaction id of the received message). If there is no DSN entry yet, then it will be created. If all entries for the particular DSN have been tried, “release” (schedule) the DSN to be sent. See also Section 2.4.6. Question: do DSN cause new entries in the main queue or do we just change the type of the recipient entry?

1. Successful delivery: Unless a SUCCESS DSN is requested the recipient can be removed from the queue and the reference counter in the sender entry is decremented. If that reference counter reaches zero, the sender entry is removed too.

2. Permanent delivery failure: If no FAILURE DSN is requested the recipient can be removed from the queue and the reference counter in the sender entry is decremented. If that reference counter reaches zero, the sender entry is removed too.

3. Temporary delivery failure: Check the timeouts for this entry:

   (a) Warning: if no DELAY DSN is requested nothing extra needs to be done.

   (b) Return: similar to permanent delivery failure (only the error code in the DSN is different).

   If the entry will be tried later on, i.e., the queue return timeout isn't reached, then determine the next time for retry. Update the entry in the deferred queue (this may require moving the entry from the incoming cache to the deferred queue).

3.4.15 Interface to AR

The AR API is briefly described in Section 3.6.3. An open question is when the QMGR should call the AR. There are at least the following possibilities:

1. for incoming mail (from SMTPS):
   (a) as soon as the recipient is received by the QMGR it can send it off to the AR to get a resolved address. This have the advantage that address resolving can be handled concurrently with mail receival. However, it might make handling of the recipient address state a little bit more complicated since we now have to keep track whether the address has been resolved and we have to decide whether an unresolved address can be moved from INCEDB to ACTEDB, in which case the returned information from the AR must be associated with the correct EDB. Additionally, the data structure used for recipient addresses in INCEDB must accommodate for resolved addresses too.
   (b) the address can be resolved when it is placed in the ACTEDB. If the address resolution takes a long time, then those entries in the ACTEDB are not really *active* and they just take up space in that EDB without being ready for scheduling.
2. for deferred mails (from DEFEDB):
   (a) the resolved addresses are stored in the DEFEDB.
   (b) the address can be resolved (again) when it is placed in the ACTEDB. The address resolution depends on data that may change over time; see Section 3.13.6. Moreover, we may not want to store all necessary delivery information in DEFEDB, but only the recipient address itself, see Section 3.4.10.6.

3.4.15.1 Aliases

The SMAR may expand aliases in which case it can return a list of (resolved) addresses. The QMGR must make sure that the new addresses are either safely stored in persistent storage or that the operation is repeatable. The simple approach is to store the new addresses in DEFEDB after they have been received from SMAR and remove the address which caused the expansion from the DB in which it is stored[11] (IBDB or DEFEDB). If the new addresses are stored only in AQ, then the handling becomes complicated due to potential delivery problems and crashes before all expanded addresses have been tried. The expansion would be done in a two step process:

1. Add all new addresses to AQ (which may fail due to space restrictions, in which case all addresses should be written to DEFEDB) and mark the original addresses as "expanded" and the new addresses as "is alias of" with a reference to the original address.
2. Try delivery for all new addresses. After all of them have been tried, mark the original addresses as "replaced".

Note: the design requires that all data in AQ can be removed (lost) at any time without losing mail.

If QMGR is terminated between step one and two, the alias expansion will be repeated the next time the (original) address is selected for delivery. If QMGR is terminated during step two, i.e., the delivery of expanded addresses, then this approach may result in multiple deliveries to the same recipient(s).

[11] Unless it is required for other purposes, e.g., logging or bounces.

Note: the current functions to update the recipient status after a delivery attempt do not yet deal with recipients resulting from an alias expansion.

For 1-1 aliases it seems simplest to replace the current address with the new one, which avoids most of the problems mentioned above[12].

In a first step 1-n ($n > 1$) alias expansion should be done by writing all data to DEFEDB. Later on optimizations can be implemented, e.g., if n is small, then the expansion is done in AQ only.

3.4.16 Updating Data in Queues after a Delivery Attempt

When a delivery attempt (see 4d in Section 2.4.3.2) has been made, the recipient must be taken care of in the appropriate way. Note that a delivery attempt may fail in different stages (see Section 2.4.3.1), and hence updating the status of a recipient can be done from different parts of QMGR:

1. The address resolver may return an error. This should happen rarely. Updating a recipient status can be a costly operation because it involves updating information on persistent storage, e.g., DEFEDB and maybe IBDB. Instead of doing this each time a recipient address resolution causes an error, the error is simply "recorded" in the recipient data structure (in AQ). Either the scheduler or a cleanup task will take care of it when it comes along such an entry. This is more likely to put several status updates together and hence they can be handled in a single transaction. Alternatively the data may be stored in request lists which are taken care of when a certain size or a timeout is exceeded.

2. A delivery agent may return an error. This is the "usual" case because hereafter the recipient status must be updated anyway.

3. An entry is too long in the active queue. This may happen if either the address resolver or the delivery agent malfunction and do not return a result. A cleanup task must take care of this.

See Section 2.4.3.4 for a description what needs to be done after a delivery attempt has been made. As mentioned there it is recommended to perform the update for a delivery attempt in one (DB) transaction to minimize the amount of I/O and to maintain consistency. To achieve this, request lists are created which contain the changes that should be made. Change requests are appended to the list when the status for recipients are updated. After all changes (for one transaction or one scheduler run) have been made, the changes are committed to persistent storage. Updates to DEFEDB are made before updates to INCEDB (as explained in Section 2.4.1), that is, first the request list for DEFEDB is committed and if that succeeded, the request list of IBDB is committed to disk.

Here's a more detailled version of the description in Section 2.4.3.4 but only for the case that no special DSN is requested, i.e., without DSN support (RFC 1894). After a delivery attempt, the recipient is removed from ACTEDB and

1. after a successful delivery attempt the recipient must be removed and the counters in the transaction context must be updated. When all references to a CDB entry have been removed, then that entry must be removed too.

    (a) If it is from INCEDB, remove the entry from INCEDB, i.e., IBDB and possibly IQDB if the entry is not removed earlier on.

[12] Again, it might be necessary to keep the original address.

   (b) If the entry is from DEFEDB, remove the recipient entry from it and update the transaction context in DEFEDB too (its counters have been changed). This should be done as one transaction to maintain consistency.

2. for a temporary delivery failure

   (a) the recipient is moved to DEFEDB if it was from INCEDB before.

   (b) otherwise the status of the recipient in DEFEDB is updated.

   If the recipient is too long in the queue, then it is treated like a permanent delivery failure (see below, item 3).

3. for a permanent delivery failure the recipient causes a DSN unless it is a double bounce, in which case the recipient is removed (after logging the fact), i.e., item 1b applies. If the recipient is already a DSN, then a double bounce is generated. Moreover,

   (a) the recipient is moved to DEFEDB if it was from INCEDB before see item 2a above.

   (b) otherwise the status of the recipient address in DEFEDB is updated.

3.4.16.1 Preserving Order of Updates

Updates to DEFEDB are partially ordered. If multiple threads prepare updates for DEFEDB, they may contain changes for the same transaction(s). The order of updates of the transaction(s) in AQ must be reflected in the order of updates in DEFEDB. There can either be a strict order or a partial order, i.e., two updates need to be done only in a certain order if they are for the same transaction. To simplify implementation, a strict order is preserved[13].

Each function that needs to update DEFEDB creates a "request list", i.e., a list of items that need to be committed to DEFEDB. This has two advantages over updating each entry individually:

1. The update is done as one transaction, i.e., atomically.

2. Only one transaction needs to be done instead of several smaller ones. Since DEFEDB changes requires disk I/O it is useful to bundle several requests into one.

3.4.16.1.1 Preserving Order of Updates: Approach 1 The first approach (more flexible and more complicated than approach 2) is to enforce just an ordering on writing data to DEFEDB by: asking for a "sequence number" and enforcing the ordering by that sequence number in the write routine.

Algorithm:

Use a mutex, a condition variable, and two counters: $first$ and $last$.

Initialize $first$ and $last$ to 0.

Invariants: $first \leq last$. Number of entries: 0 if $first = last = 0$ otherwise $last - first + 1$, i.e., there are no requests iff $first = last = 0$.

Resetting $first$ and $last$ to 0 is done to minimize the chance for an overflow.

Get an entry:

[13] For now; this can be relaxed later on, which complicates the algorithm however. Hence it should be determined first whether strict ordering causes a bottleneck.

```
lock(mutex)
if (first == 0)
  first = last = 1
  n = 1
else
  n = ++last
unlock(mutex)
return n
```

To write a request list:

```
lock(mutex)
while (number != first)
  cond_wait(cond, mutex)
unlock(mutex)

lock(write_mutex)
... write list ...
unlock(write_mutex)

lock(mutex)
assert(number == first)
assert(first <= last)
if (first < last)
  ++first
  signal(cond)
else
  first = last = 0
unlock(mutex)
```

Note: there is currently no absolute prevention of overflowing *last*. If this needs to be done, then *last* would be checked in the function that increases it and if it hits an upper limit, it would wait on a condition variable. The signalling would be done in the function that increases first: if it reaches the upper limit, then it would signal the waiting process.

3.4.16.1.2 Preserving Order of Updates: Approach 2 A simpler way to solve the problem is to lock DEFEDB and AQ, and write changes to DEFEDB before it is unlocked. Even though that keeps DEFEDB locked while making changes to AQ (which prevents only a reader process from accessing DEFEDB even though it might be possible to allow that otherwise), this seems like the simplest approach to solve the problem.

3.4.17 Load Control Functionality

According to Section 2.4.7 QMGR must control the local load of the system, Section 3.4.10.13 describes the possible data structures for this purpose. In this section the functionality will be specified.

An MTS has some modules that produce data (SMTP servers) and some which consume data (SMTP clients, delivery agents in general). The simplest approach to control how much storage is needed is to regulate the producers such that they do not exceed the available storage capacities. To accomplish this two thresholds are introduced for each resource:

1. an upper threshold T_u: if this is exceeded then the producers are throttled or stopped,

2. a lower threshold T_l: if the actual value falls below the lower threshold then the producers are enabled again.

To allow for a fine grained control the capacity C (range: $0 \dots 100$) of the producers should be a regulated on a sliding scale proportional to the actual value U if it is between the lower and upper threshold. Capacity C is the *inverse* of the resource usage R, i.e., $C = 100 - R$.

if $U \geq T_u$ then $C := 0$ else if $U \leq T_l$ then $C := 100$ else $C := 100 * (T_u - U)/(T_u - T_l)$

or

if $U \geq T_u$ then $R := 100$ else if $U \leq T_l$ then $R := 0$ else $R := 100 * (U - T_l)/(T_u - T_l)$

For multiple values a loop can be used which is stopped as soon as one value exceeds its upper threshold. Computing the capacity can be done as follows:

$S := 0; C := 100; N := 0$

for each U do
if $U \geq T_u(U)$ then $C := 0$; break;
else if $U \leq T_l(U)$ then $C := 100$;
else $C := 100 * (T_u(U) - U)/(T_u(U) - T_l(U))$;
$C := S + C; N := N + 1$
done
if $C > 0$ then $C := S/N$

Computing the resource usage is done in the corresponding way:

$S := 0; R := 0; N := 0$

for each U do
if $U \geq T_u(U)$ then $R := 100$; break;
else if $U > T_l(U)$ then $S := S + 100 * (U - T_l(U))/(T_u(U) - T_l(U))$;
$N := N + 1$
done
if $R < 100$ then $R := S/N$

Notes:

1. The second algorithm is slightly optimized: if $U \leq T_l(U)$ then R is zero, hence it doesn't need to be added to the sum S.

2. these algorithms compute the arithmetic mean whereas the geometric mean gives better results. However, computing the latter requires floating point arithmetic (it's the n-th cube of the product of the values) and this does not seem to be worth the effort.

3.4.17.1 Load Control Functionality: Throttling

In general, whenever a resource is used, it must be checked whether it is exhausted. For example, whenever an item is added to a DB the result value is checked. If the result is an error which states that the DB is full, the SMTP servers (producers) must be stopped. This is the simple case where the usage of a resource reaches the upper limit (exceed the upper threshold).

After several operations have been performed of which some contain additions to DBs, a generic throttle function can be called which checks whether any of the resource usages exceeds its lower limit in which case the SMTP servers are throttle accordingly. This needs to be done only if the new resource usage is sufficiently different from the old value, otherwise it is not worth to notify the producers of a change.

3.4.17.2 Load Control Functionality: Unthrottling

If the system is in an overloaded state, i.e., the producers are throttled or even stopped, then the resource usage must be checked whenever items are removed from DBs. If the new resource usage is sufficiently less than the old value, the SMTP servers are informed about that change (*unthrottled*). Alternatively, the producers can be unthrottled only after all resource usages are below their lower thresholds.

3.4.17.3 Handling out of Memory

Unfortunately there is no simple, portable way to determine the amount of memory that is used by a process. Moreover, even though `malloc(2)` is supposed to return `ENOMEM` if there is no more space available, this does not work on all OS because they overallocate memory, i.e., they allocate memory and detect a memory shortage only if the memory is actually used in which case some OSs even start killing processes to deal with the problem. This is completely unacceptable for robust programming: why should a programmer invest so much time in resource control if the OS just “randomly” kills processes? One way to deal with this might be to use `setrlimit(2)` to artificially limit the amount of memory that a process can use, in which case `malloc(2)` should fail properly with `ENOMEM`.

If the OS properly returns an appropriate error code if memory allocation fails, then the system can be throttled as described in 3.4.17.1. However, it is hard to recover from this resource shortage because the actual usage is unknown. One way to deal with the problem is to reduce the size of the memory caches if the system runs out of memory. That can be done in two steps:

1. try to remove some items from memory caches which are not really necessary for proper operation, and which do not require updating persistent caches. For example, entries in the active queue that have not yet been scheduled and are from deferred queue can be removed without having to update the deferred queue[14].
2. (temporarily) decrease the upper limits for memory caches to reduce the amount of memory that can be used before throttling sets in.

Now the system can either stay in this state or after a while the limits can be increased again. In the latter case the resource control mechanisms may trigger again if the system runs out of memory again. It might be useful to keep track of those resource problems to adapt the resource limits to avoid large variations and hence repeated operation of the throttling/unthrottling code (i.e., implement some “dampening” algorithm).

3.4.18 Manual Interaction with QMGR

There are two different requirements about interacting with the queue manager:

1. Getting status information.

[14] *Question*: does it require updating EDBC?

2. Triggering certain actions that are usually done internally based on the configuration, e.g., performing a queue run for a certain recipient.

For both forms of interaction a control socket can be used. However, in some cases it might be useful to give different access rights to the two interactions: reading status information might be allowed for more persons than requesting actions.

For interaction with QMGR the same basic communication protocol (RCBs) will be used to simplifiy (and unify) the implementation. For simplicity it seems to be useful to allow only one control connection at a time.

3.4.18.1 Getting Status Informations

The QMGR should be able to list almost all of its internal status information when requested. In some cases it might be useful to ask (and return) only a subset of the data to reduce the amount of data that is sent.

3.4.18.2 Triggering Actions

Available actions should include:

1. Schedule a recipient (based on its id) for delivery "as soon as possible".
2. Remove a recipient (based on its id) from the queue (discard).
3. Remove a transaction (based on its id) from the queue (discard).

3.5 SMTP Server Daemon

3.5.1 External Interfaces

The external interface of the SMTP server (also called smtpd in this document) is of course defined by (E)SMTP as specified in RFC 2821. In addition to that basic protocol, the sendmail X SMTP server will support: RFC 1123, RFC 1869, RFC 1652, RFC 1870, RFC 1891, RFC 1893, RFC 1894, RFC 1985, RFC 2034, RFC 2487, RFC 2554, RFC 2852, RFC 2920. See Section 1.1.1 for details about these RFCs.

Todo: verify this list, add CHUNKING, maybe (future extension, check whether design allows for this) RFC 1845 (SMTP Service Extension for Checkpoint/Restart).

3.5.2 Control Flow

A SMTP session may consist of several SMTP transactions. The SMTP server uses data structures that closely follow this model, i.e., a session context and a transaction context. A session context contains (a pointer to) a transaction context, which in turn contains a pointer back to the sessions context. The latter "inherits" its environment from the former. The session context may be a child of a daemon context that provides general configuration information. The session context contains for example information about the sending host (the client) and possibly active security and authentication layers.

The transaction context contains a sender address (reverse-path buffer in RFC 2821 lingo), a list of recipient addresses (forward-path buffer in RFC 2821 lingo), and a mail data buffer.

3.5.2.1 Startup

At startup a SMTP server registers with the QMGR. It opens a communication channel to the QMGR and announces its presence. The initial data includes a least a unique identifier for the SMTP server which will be used later on during communication. Even though the identification of the communication channel itself may be used as unique identifier, e.g., a file descriptor inside the QMGR, it is better to be independent of such implementation detail. The unique identifier is generated by the MCP and transferred to the SMTP server. It may also act as a "key" for the communication between SMTPS and QMGR/MCP if the communication channel could be abused by other programs. In that case, the MCP tells the QMGR about the new SMTPS (including its key).

3.5.2.2 States

It seems appropriate to have two different states: the state of the session and that of the transaction, instead of combining these into one. Not all of the combinations are possible (if there is no active session, there can't be a transaction). Many of the states may have substates with simple (enumerative) or complex (e.g., STARTTLS information) descriptions, some of them must be combinable ("additive"). Question: How to describe "additive" states? Simplest way: bit flags, binary or. So: can we put those into bitfields?

Session states:

1. SMTPS-SESSION-NONE: no session active
2. SMTPS-SESSION-CONNECTING: connection attempt
3. SMTPS-SESSION-CONN-INIT: connection succeeded, 220 greeting given.
4. SMTPS-SESSION-EHLO: EHLO command received; aborts a transaction (RFC 2821, 4.1.4 [Kle01]). Discard transaction context, new transaction state is SMTPS-TA-NONE.
5. SMTPS-SESSION-HELO: HELO command received; aborts a transaction (RFC 2821, 4.1.4 [Kle01]). Discard transaction context, new transaction state is SMTPS-TA-NONE.
6. SMTPS-SESSION-AUTH: AUTH command received (can only be given once, is additive).
7. SMTPS-SESSION-STARTTLS: STARTTLS command received (can only be given once, is additive).
8. SMTPS-SESSION-RSET: RSET command received. Discard transaction context, new transaction state is SMTPS-TA-NONE. Question: is this a session or a transaction command?
9. SMTPS-SESSION-QUIT: QUIT command received
10. SMTPS-SESSION-NULL: reject almost all commands, i.e., act as "nullserver".

Transaction states:

1. SMTPS-TA-NONE: no transaction active.

2. SMTPS-TA-INIT: transaction data initialized.
3. SMTPS-TA-MAIL: MAIL command has been given. New mail transaction starts, create new transaction context. Check syntax of address and extensions as well as whether the requested extensions are available and valid.

   Multiple MAIL commands are not allowed during a transaction, i.e., MAIL is only valid if state is SMTPS-TA-INIT.
4. SMTPS-TA-RCPT: RCPT command has been given. Add recipient to transaction context (list of recipients). Check syntax of address and extensions as well as whether the requested extensions are available and valid. Also perform anti-spam checks, esp. anti-relay.

   RCPT is only valid if state is SMTPS-TA-MAIL or SMTPS-TA-RCPT.
5. SMTPS-TA-DATA: DATA command has been given. Collect data. DATA is only valid if state is SMTPS-TA-RCPT.
6. SMTPS-TA-DOT: Final dot of DATA section has been given (this is a transient state). Commit mail content to stable storage or deliver it. Then acknowledge receipt and clear transaction (buffers), i.e., essentially delete the transaction context in the SMTP server. Next state is SMTPS-TA-NONE or SMTPS-TA-INIT.

Some of these states require a certain sequence, others don't. We can either put this directly into the code (each function checks itself whether its prerequisites are fulfilled etc) or into a central state-table which describes the valid state changes (and includes appropriate error messages). It seems easier to have this coded into the functions than having it in the event loop which requires some state transition table, see libmilter.

VRFY, EXPN, NOOP, and HELP can be issued at (almost) any time and do not modify the transaction context (no change of buffers).

3.5.2.3 Data Structures

There need to be at least two data structures in the SMTP server; one for a transaction, one for a session.

Additionally, a server context is used in which configuration data is stored. This context holds the data that is usually stored in global variables. In some cases it might be useful to change the context based on the configuration, e.g., per daemon options. Hence global variables should be avoided; the data is stored instead in a context which can be easily passed to subroutines. The session context contains a pointer to the currently used server context.

Session:

session-id	session identifier, maybe obtained from QMGR
connect-info	identification of connecting host (IP address, host name, ident)
fd	file descriptor/handle for connection
helo-host	host name from HELO/EHLO
access times	start time, last read, last write
status	EHLO/HELO, AUTH, STARTTLS (see above)
features	features offered: AUTH, TLS, EXPN, ...
workarounds	work around bugs in client
reject-msg	message to use for rejections (nullserver)
auth	AUTH context
starttls	TLS context
transaction	pointer to current transaction

Transaction:

transaction-id	transaction identifier, maybe obtained from QMGR
mail	address, arguments (decoded?)
rcpt-list	addresses, arguments (decoded?)
cdb-id	CDB identifier (obtained from cdb?)
cmd-failures	number of failures for certain commands
session	pointer to session

Question: How much of the transaction do we need to store in the SMTP server? The QMGR holds the important data for delivery. The SMTP server needs the data to deal with the current session/transaction.

3.5.2.4 Detailled Control Flow

Todo: describe a complete transaction here including the interaction with other components, esp. queue manager.

Notice: this probably occurs in an event loop. So all functions are scheduled via events (unless noted otherwise).

As described in Section 3.16.4 the I/O layer will try to present only complete records to the middle layer. However, this may be intertwined with the event loop because otherwise we have a problem with non-blocking I/O. For the purpose of this description, we omit this I/O layer part and assume we receive full records, i.e., for the ESMTP dialogue CRLF terminated lines.

1. The SMTP server waits for incoming connection attempts.

   Before accepting any new connection, a session context is created. This allows the server to react faster (it doesn't need to create the context on demand), and it is guaranteed that a session context is actually available. Otherwise the server may have not enough memory to allocate the session context and hence to start the session.

   smtps_session_new(OUT smtps-session-ctx, OUT status): create a new session context.

2. The SMTP server receives an incoming connection attempt.

   smtps_session_init(IN fd, IN connect-info, INOUT smtps-session-ctx, OUT status): Initialize session context, including state of session, fill in appropriate data.

   The basic information about the client is available via a system call (IP address). The (canonical) host name can be determined by another system call (gethostbyaddr(), which may be slow due to DNS lookups). Question: do we want to perform this call in the SMTP server, in the QMGR, or in the AR? There should be an option to turn off this lookup completely and hence only rely on IP addresses in checks. This can avoid those (slow) DNS lookups. SMTPS optionally (configurable) performs an auth (ident) lookup.

   smtps_session_chk(INOUT smtps-session-ctx, OUT status): SMTPS contacts the queue manager and milters that are registered for this with the available data (IP address, auth result). The queue manager and active milters decide whether to accept or reject the connection. In the latter case the status of the server is changed appropriately and most commands are rejected. In the former case a session id is returned by the queue manager. Further policy decisions can be made, e.g., which features to offer to the client: allow ETRN, AUTH (different mechanisms?), STARTTLS (different certs?), EXPN, VRFY, etc, which name to display for the 220 greeting, etc. The QMGR optionally logs the connection data.

   Question: how can we overlap some OS calls, i.e., auth lookup, gethostbyaddr(), etc?

Output initial greeting (usually 220, may be 554 or even 421). Set timeout. Progress state or terminate session.

3. Read SMTP command from client. In case of timeout, goto error treatment (abort session). Call the corresponding SMTP function.
4. If SMTP commands are used that change the status of a session (e.g., STARTTLS, AUTH), those are executed and their effects are stored in the session context. Those functions are:

   smtps_starttls(INOUT session-ctx, OUT state)

   smtps_auth(INOUT session-ctx, OUT state)
5. For each transaction a new envelope is created and the commands are communicated to the queue manager and the address resolver for validation and information. Other processes (esp. milters) might be involved too and the commands are either accepted or rejected based on the feedback from all involved processes.
6. When an e-mail is received (final dot), the queue manager and the SMTP server must either write the necessary information to stable storage or a delivery agent must take over and deliver the e-mail immediately. The final dot is only acknowledged after either of these actions completed successfully.

All the following functions check whether their corresponding commands are allowed in the current state as stored in the session/transaction context. Moreover, they check whether the arguments are syntactically correct and allowed depending on the features in the session context. The server must check for abuse, e.g., too many wrong commands, and act accordingly, i.e., slow down or in the worst case closing the connection. Functions in SMTP server (see Section 3.4.12 for counterparts in the QMGR):

smtps_helo(INOUT session-ctx, IN line, OUT state): store host name in session context, clear transaction context, reply appropriately (default 250).

smtps_ehlo(INOUT session-ctx, IN line, OUT state): store host name in session context, clear transaction context, reply with list of available features.

smtps_noop(INOUT session-ctx, IN line, OUT state): reply appropriately (default 250).

smtps_rset(INOUT session-ctx, IN line, OUT state): clear transaction context, reply appropriately (default 250).

smtps_vrfy(INOUT session-ctx, IN line, OUT state): maybe try to verify address: ask address resolver.

smtps_expn(INOUT session-ctx, IN line, OUT state): maybe try to expand address: ask address resolver.

smtps_mail(INOUT session-ctx, IN line, OUT state): start new transaction, set sender address.

smtps_rcpt(INOUT session-ctx, IN line, OUT state): add recipient to list.

smtps_data(INOUT session-ctx, IN line, OUT state): start data section (get cdb-id now or earlier?).

smtps_body(INOUT session-ctx, IN buffer, OUT state): called for each chunk of the body. It's not clear yet whether this needs to be line oriented or whether it can receive simply entire body chunks. It may have to be line oriented for the header while the rest can be just buffers, unless we want to perform operations on the body in the SMTP server too. This function is responsible for recognizing the final dot and to act accordingly.

Question: how do we deal with information sent to the QMGR and the responses? Question: how do we describe this properly without defining the actual implementation, i.e., with leaving ourselves room for possible (different) implementations?

Assuming that the SMTP servers communicate with the QMGR, we need some asynchronous interface. The event loop in the SMTP servers must then also react on data sent back from the QMGR to the SMTP servers. As described in Section 3.5.2.5, events for the session must be disabled or ignored while the SMTP server is "waiting" for a response from the QMGR.

If the QMGR is unavailable the session must be aborted (421).

3.5.2.4.1 PIPELINING: Concurrency There is one easy way to deal with PIPELINING in the SMTP server and one more complicated way. The easy way is to more or less ignore it, i.e., perform the SMTP dialogue sequentially and let the I/O buffering take care of PIPELINING. This works fine and is the current approach in sendmail V8. The more complicated way is to actually trying to process several SMTP commands concurrently. This can be used to "hide" latencies, e.g., if the address resolver or the anti-spam checks take a lot of time – maybe due to DNS lookups – then performing several of those tasks concurrently will speed up the overall SMTP dialogue. This requires of course that some of those lookups can be performed concurrently as it is the case if an asynchronous DNS resolver is used.

3.5.2.4.2 ID Generation Question: who generates the ids (session-id, transaction-id)? The QMGR could do this because it has the global state and could use a running counter based on which the ids are generated. This counter would be initialized at startup based on the EDB, which would have the last used value. However, it might reduce the amount of communication between SMTPS and QMGR if the former could generate the ids themselves. For example, just asking the QMGR for a new id when a session is initialized is a problem: which id should a SMTPS use to identify this request? The ids must be unique for the "life time" of an e-mail, they should be unique for much longer. Using a timestamp (as sendmail 8.12 does more or less) causes problems if the time is set backwards. Hence we need a monotonically increasing value. The ids should have a fixed length which makes several things simpler (even if it is just the output formatting of mailq). Since there can be several SMTPS processes, a single value in a shared location would require locking (the QMGR could provide this as discussed above), hence it would be more useful to separate the possible values, e.g., by adding a "SMTPS id" (which shouldn't be the pid) to the counter. This SMTPS id can be assigned at startup by the MCP. It seems tempting to make the transaction id name space disjunct from the session id name space and to provide a simple relation between those. For example, a session id could be: "S *SMTPS-Id counter*" and a transaction id could be: "T *SMTPS-Id counter ta-counter*". However, this makes it harder to generate fixed-length ids, unless we restrict the number of transactions in a session and the number of SMTPS processes. If we do that, then we must not set those limits to small (maybe 9999 would be ok), which on the other hand wastes digits for the normal cases (less than 10 for both values). Using 16 bit for the SMTPS id and 64 bit for the counter results in a string of 20 characters if we use base 16 encoding (hex). We could go to 16 (14) characters if we use base 32 (64) encoding. However, if we use base 64 encoding, the resulting id can't be used as filename on NT since it doesn't use case sensitive filenames (*Check this*).

3.5.2.5 Pipelining

If an SMTP server is implemented as an event-driven state machine, then a small precaution must be taken for pipelining. After an SMTP command has been read and a thread takes care of processing it, another SMTP command may be available on the input stream, however, it should not be processed before the processing of the current command has been finished. This can either be achieved by disabling the I/O events for that command stream during processing, or the session must be locked and any data available must be buffered while processing is active. It is not clear yet which of those two approaches is better; changing the set of I/O events may be an expensive operation; additional buffering of input data seems to be superfluous since the OS or the I/O layer takes care of that and more buffering will complicate the I/O handling (and violates layering). Maybe the loops which checks for I/O events should check only those sessions which are not active, i.e., which are currently not being processed. However, then the mechanism that is used to check for I/O events must present that I/O event each time (select() and poll() do this, how about others?).

3.5.2.6 Anti-Spam Checks

Note: there are different checks all of which are currently called "anti-spam checks". Basically they can be divided into two categories:

1. relay checks;
2. other policy checks.

Usually those two are merged together for simplicity. The basic assumption here is that a system which is allowed to relay does not need to undergo other policy checks, i.e., it is *trusted*. For each step in an ESMTP session/transaction, various tests can be applied which may have an immediate or delayed effect. For example, a client can be rejected at connection time (immediate) or it may be authorized to relay (delayed until `RCPT`). In most cases, a test returns a *weighted* result:

1. `REJECT` (error): reject command (permanent or temporary error).
2. `NEUTRAL` (continue): proceed (maybe apply subsequent tests for this command/stage); this is the default if nothing else is found/returned.
3. `OK` (accept): accept (skip subsequent tests for this command/stage).
4. `RELAY`: accept and allow relaying (skip subsequent tests for this command/stage).

Here `RELAY` is *stronger* than `OK` because it grants additional rights. However, it is not clear whether `RELAY` should override also tests in other (later) stages or just a possible anti-relay test (which might be done for `RCPT`). However, there might be cases where this simple model is not sufficient. Example: B is a backup server for A, but B does not have a complete list of valid addresses for A. Hence it allows relaying for a (possible) subset of addresses and temporarily rejects other (unknown) addresses, e.g.,

```
To:a1@A   RELAY
To:a2@A   RELAY
To:@A     error:451 4.7.1 Try A directly
```

If session relaying actually overrides all subsequent tests, then this example works as expected, because hosts which are authorized to relay can send mail to A. However, if an admininstrator wants to maintain blacklists which are not supposed to be used then relaying tests and other policy tests need to be separate. An alternative solution would be using the QUICK: modifier (see below about QUICK) to indicate that also positive results (OK, RELAY) should have an "immediate" effect, i.e., they override later rules. This seems to be consistent with the way it is used in firewall applications ([pf]).

The order (and effect) of checks must be specifyable by the user; in sm8 this is controlled by the `delay_checks` feature. This allows only for either the "normal" order (connect, MAIL, RCPT) or the reverse order (RCPT, MAIL, connect). With the default implementation this means that MAIL and connect checks are done for each RCPT which can be very inefficient.

3.5.2.6.1 Combining Anti-Spam Checks The implementation of the anti-spam checks is non-trivial since multiple tests can be performed and hence the individual return values must be combined into one. As explained in Section 3.14.1, map lookups return two values: the lookup result and the RHS if the key was found. For example, for anti-relay checks the interesting results are (written as pair: (lookup-result, RHS), where RHS is only valid if lookup-result is FOUND):

- (FOUND, RELAY): allow relaying, no further tests required,
- (NOTFOUND, -) and (PERMFAIL, -) are treated as NO-DECISION-YET (or CONTINUE): if there are more tests: run those, otherwise deny or temporary failure,
- (TEMPFAIL, -): if there are more tests: run those, otherwise temporary failure.

The last case will be carried through unless an allowance is given. This is fairly simple and can be coded into the binary:

```
state = NO-DECISION-YET;
while (state != RELAY && more(tests)) {
    (r, rhs) = test();
    switch (r, rhs) {
      case (TEMPFAIL, -):        state = r;         break
      case (FOUND, RELAY):       return r;
      default:                                      break;
   }
}
if (state == NO-DECISION-YET)      return REJECT;
else                               return state;
```

These anti-relay checks should be done in several phases of the ESMTP dialogue.

1. Session: Is the client authorized to relay? If so, set a flag in the session state. Note: the authorization can be done based on the client IP address or be the result of some other authentication such as STARTTLS or AUTH. If the client authentication causes a temporary failure, then this must be stored in the session state too (see above) to give the correct error message later on (temporary failure instead of a permanent error).
2. Transaction: Theoretically relaying could also be allowed per MAIL address. This is insecure and should not be implemented by default, however, an extension module may want to do this.
3. Recipient: If relaying is not allowed per session, then each RCPT must be checked. This can be extended to allow only certain clients or senders to reach specific recipient addresses, which is not directly related to anti-relaying however.

To do this, the flow control depicted above is enhanced such that the state is initialized properly by the state of the surrounding ESMTP phase, i.e., for session it is initialized to NO-DECISION-YET, for RCPT it is set to the relay-state of the session.

It becomes more complicated for other anti-spam tests. For example: one test may return TEMPFAIL, another may return OK, a third one REJECT, a fourth one REJECT but with a 4xy error. Question: how to prioritize? Question: should it be just a fixed algorithm in the binary or something user-definable? Maybe compare `nsswitch.conf(2)`([nss]) which provides a "what to do in case of..." decision between different sources (checks in the anti-spam case)? That is:

Result	Meaning
FOUND	Requested database entry was found
NOTFOUND	Source responded ”no such entry”
PERMFAIL	Source is not responding or corrupted (permanent error)
TEMPFAIL	Source is busy, might respond to retries (temporary error)

Action	Meaning
CONTINUE	Try the next source in the list
RETURN	Return now

```
<entry>     ::= <database> ":" [<source> [<criteria>]]*
<criteria>  ::= "[" <criterion>+ "]"
<criterion> ::= <status> "=" <action>
<status>    ::= "FOUND" | "NOTFOUND" | "PERMFAIL" | "TEMPFAIL"
<action>    ::= "RETURN"  | "CONTINUE"
<source>    ::= "file"  | "DNS" | ...
```

So we need a state that is carried through and can be modified, and an action (break/return or continue).

```
<entry>      ::= <what> ":" [<test> [<criteria>]]* <return>
<test>       ::= some-test-to-perform
<criteria>   ::= "[" <criterion>+ "]"
<criterion>  ::= <status> "?" [<assignment>] <action> ";"
<status>     ::= "FOUND" | "NOTFOUND" | "PERMFAIL" | "TEMPFAIL"
<action>     ::= <return> | "continue"
<return>     ::= "RETURN" [<value>]
<assignment> ::= "state" "=" <value> ","
<value>      ::= <status> | "status"
```

Alternatively we can use a marker that says: “use this result” (e.g., pf([pf]): `quick`); by default the last result would be used. However, this doesn't seem to solve the TEMPFAIL problem: the algorithm needs to remember that there was a TEMPFAIL and return that in certain cases.

Question: can we define an algebra for this? Define the elements and the operations such that a state and a lookup result can be combined into a new state? Would that make sense at all? Would that be flexible enough?

3.5.2.6.2 Requirements for Anti-Spam Checks *Question*: what are the requirements for anti-spam? Are the features provided by sm8 sufficient? Requirements:

1. override builtin checks.
2. reject connections, sender and recipient address.
3. allow connections or sender address based on recipient, i.e., override rejections. Note: currently this is a simple all or nothing scheme, i.e., if a recipient address is marked as “spam-friend”, then all checks are turned off. A more complex configuration would selectively turn on/off checks [Krü98]. This is done by returning a list of identifiers which define the set of checks to enable (or disable). Using a map for this is an example of “configuration via map” which is not a good idea for anything more complex than returning a simple (single) value.

Question: Where do we put the anti-spam checks? Some of them need similar routines as provided by the address resolver, e.g., for anti-relay we need the recipient address in a canonical form, i.e., which is the real address to which the mail should be sent? Maybe we need an interface that allows different modes, i.e., one for verification etc only, one for expansion.

3.5.2.6.3 Current Behavior of Anti-Spam Checks For each stage in the SMTP dialogue there are builtin checks (which usually can be turned on/off via an option, e.g., accept-unresolvable-domain), and there are checks against the access map. The order of checks specifies whether a check can override other ("later") checks unless special measures are taken (see below). Note: some of the builtin checks are configured by the user, hence it does not make sense to provide another option to override these because that can be done by setting the option accordingly, e.g., for connection rate it simply requires increasing the limit.

- Connect: builtin: rejection based on number of open connections, number of connections within the last time interval, etc.

  user defined: check host name and address against maps (access map, DNS based blacklists).
- EHLO: currently not really supported.
- MAIL:

  builtin: syntax checks[15].

  user defined: check address against maps (access map, DNS based blacklists).

  builtin: unresolvable domain, others?
- RCPT:

  builtin: syntax checks, local user: does the address exist?

  user defined: check address against maps,

  builtin (configurable via map): check anti-relay.
- DATA: currently not really supported.

As usual these checks may all be done during the RCPT stage or when they occur during the SMTP dialogue.

Before the control flow of sm8 is described, the values (data, RHS) of a map lookup need to be explained:

1. REJECT: return permanent error; this is just an abbreviation for ERROR:550 error-text.
2. ERROR:error-code-text return permanent or temporary error. Maybe this should be split in two values: TEMP and PERM, otherwise it is necessary to inspect the error code.
3. CONTINUE (default): this is the value that is used if nothing is found, it currently is not a valid RHS. This is basically the same as SKIP; which can be used to exempt a subnet/subdomain if a net/domain is listed, e.g., reject all of 10.* but 10.1.2.*. CONTINUE indicates that this particular test did not produce a useable result (a match), but further tests can (should) occur (in contrast to OK).
4. OK: override other checks for the same stage.

[15] It might make sense to specify a "strictness" level for the syntax checks.

5. ACCEPT: override all further checks (maybe QUICK:OK, see below about QUICK), e.g., if a connection comes from a trusted host, accept the mail even if the sender address would be rejected due to later checks. This could be used to turn off anti-spam checks based on client address or maybe even sender address. Note: see the next item how different tags are required for recipient addresses to override anti-spam checks.

6. RELAY: allow relaying. Note: this is not really a value that is related to the anti-spam checks. Hence it causes problems, because there must be a way to distinguish between the anti-spam and the anti-relay entries in the map. This is exemplified by the use of different tags (in sendmail 8) to allow relaying to domains and to exempt domains from anti-spam tests when the feature delay-checks is activated: for the former the tag `To:` is used, for the latter its `Spam:`.

7. DISCARD: discard mail (pretend to accept it but throw it away). Note: for recipients it might be useful to have to different versions: one to discard just the recipient, another one to discard the entire transaction.

8. QUARANTINE: quarantine mail, i.e., accept it and keep in mail queue until explicitly released.

Control flow in sm8:

1. without delay-checks:
   - connect: client host name, IP address: OK (override other checks for connect), DISCARD (everything in this session), REJECT, RELAY (not used here, will be checked during RCPT again).
   - MAIL: mail sender address: OK (override other checks for MAIL), DISCARD (transaction), REJECT (even RELAY if special configuration is enabled).
   - RCPT: recipient address: OK (override other checks for RCPT), DISCARD (transaction[16]), REJECT, RELAY. If the lookup returns neither REJECT nor RELAY then the usual anti-relay tests are performed.
2. with delay-checks: all checks are done during RCPT:
   - recipient address: OK (disable other checks for RCPT), REJECT, RELAY. If the lookup returns neither REJECT nor RELAY then the usual anti-relay tests are performed. Note: disabling all other checks is not done by having the usual lookup (`To:recipient@address`), but by a different tag and a default selection; the tag is `Spam:` and the RHS defines whether further checks should be performed[17]. If yes, those checks are done in the order MAIL and then connect; otherwise the recipient address is accepted.
   - mail sender address: OK (override other checks for MAIL, don't run connect checks), REJECT (even RELAY if special configuration is enabled).
   - client host name, IP address: OK (override other checks for connect), REJECT, RELAY (not used here, has been checked at the begin of the sequence, i.e., during RCPT).

These two methods could be combined by keeping track of a state. If delay-checks is used, then MAIL and connect checks are performed for each recipient (unless overridden via an accept return code). Moreover, either the return codes could be more fine grained, e.g., for connect checks a return value could say: reject now, don't bother to do any further checks later on (compare firewall configurations ([pf]): `quick`). This can be achieved by either using different tags for the LHS or different tags for the RHS.

[16] sendmail 8 does not have DISCARD per recipient?
[17] see item 6 in the list of valid RHS above for an explanation.

- Different tags for the LHS: the tag `ClientAddr:` for the LHS would mean that in case of a match the result is used during connect, while the tag `Connect:` for the LHS would mean the RHS is used after a RCPT check. Tagging the LHS requires double the number of lookups however, hence tagging the RHS would be better, but that only works if there is at most one result. In some cases two different values are required, e.g., an entire recipient domain might be a *spamfriend* and relaying to it might be allowed (see also item 6 above). Hence this must be decided on a case by case basis.

- Different tags for the RHS: the values for the RHS are listed above. These could be expanded by an optional QUICK, e.g., QUICK:REJECT, QUICK:ERROR:error.

Question: what would the state look like? Is it possible to define an order on the value of the RHS (in that case the state would be the "maximum" of the values)?

Note: a simple state might not be good enough, e.g., if MAIL should be rejected only if RCPT matches an entry in a map, then a state for MAIL does not help. However, such a test would have to be performed in RCPT itself (and use MAIL too for its check).

Other hooks should be the same as in sendmail 8, i.e., for the various SMTP commands (DATA, ETRN, VRFY, EXPN) and some checks for TLS or AUTH.

3.5.2.6.4 Note About Delay-Checks sendmail 8 actually delays checks until the recipient stage. This has the disadvantage that all checks are run for every recipient unless the recipient is a "spam friend". However, this also has one advantage: if all recipients for a transaction are "spam friends" then none of the other checks need to be performed. This can be an advantage if time consuming checks (e.g., DNS blacklists) are specified. It would be possible to implement this behavior in sendmail X without its disadvantage: the tests need only be performed once and the results will be cached just like they are now. However, this has one disadvantage: the `QUICK:` modifier does not work if this implementation is chosen. It might be possibly to make delay-checks an option with more states than on and off, e.g., off, delay-rejections: perform tests at corresponding SMTP stage but delay possible rejections, delay-checks: perform tests only at RCPT stage. This might be a future enhancement but it is not worth the effort for sendmail X.0.

3.5.2.7 Anti-Spam Checks: Functionality

As the experiences with sendmail 8 anti-spam checks have shown, it will be most likely not sufficient to provide some simple, restricted grammar. Many people want expansions that are only available in sendmail 8 due to the ruleset rewrite engine (even though that has several problems by itself since it was intended for address rewriting, not as a general programming language). Question: how can sendmail X provide a flexible description language for anti-spam checks (and other extensions, e.g., the address resolver)? Of course the simplest way for the implementors is to let users hack the C source code. If there is a clean API and a module concept (see also Section 3.17) then this might be not as bad as it sounds. However, letting user supplied C code interact with the sendmail source code may screw up the reliability of sendmail. A separation like libmilter is cleaner but slower.

3.5.2.8 Anti-Spam Checks: Functionality for first Release

For the first release (sendmail X.0) some basic anti-spam functionality must be provided that is sufficient for most cases. For each step SMAR returns up to three values:

1. the lookup results (SUCCESS (FOUND), NOTFOUND, TEMPFAIL, PERMFAIL). If the result is TEMPFAIL or PERMFAIL then the system should have a default reply code (which could be different for each test). For now TEMPFAIL and PERMFAIL are ignored, i.e., the default reply code is CONTINUE.

2. if the result was SUCCESS, then it returns an SMTP reply code

RELAY	100	Allow relaying
OK	250	Accept command, no further tests in this stage
CONTINUE	300	Run further tests
SSD	421	Service shutting down
TEMP	450	Temporary error
REJECT	550	Permanent error

   This code is modified by adding a constant to it if the RHS string contained QUICK[18].

   Enhancements for later releases may be:

DISCARD	600	Accept mail but discard it
QUARANTINE	700	Quarantine mail

3. if the result was SUCCESS an optional string can be returned which contains the SMTP reply code, a DSN code, and an error text.

Below is a proposal for the basic anti-spam algorithm. There are a few configuration options for this: feature delay-checks (same behavior as in sendmail 8, i.e., returning rejections is delayed to the RCPT stage).

- connect: client host name, IP address:
  - RELAY: allow relaying, store this information in the session state; override other checks for connect, e.g., potential DNS BL lookups.
  - QUICK:RELAY: allow relaying, store this information in the session state; override all other checks.
  - OK: override other checks for connect, e.g., potential DNS BL lookups.
  - QUICK:OK: override all other checks except for relay tests.
  - QUICK:REJECT: reject session.
  - REJECT: reject session unless delay-checks is set.
  - TEMP: tempfail session unless delay-checks is set.
  - CONTINUE: continue with other checks.

  Any other result is stored in the session state, i.e., the triple (result, RHS, text).

- MAIL: lookup mail sender address with tag `From:` (or `Mail:`?):
  - OK: override other checks for MAIL.
  - QUICK:REJECT: reject transaction.
  - REJECT: reject transaction unless delay-checks is set.
  - TEMP: tempfail transaction unless delay-checks is set.
  - CONTINUE: continue with other checks.

[18]It is also possible to use a negative value, but that might cause confusion with sendmail X error values.

Any other result is stored in the transaction state.

- RCPT: recipient address: first perform lookups with the tag `To:` (or `Rcpt:`?):
  - RELAY: accept recipient, run other tests if feature delay-checks is turned on.
  - QUICK:RELAY: accept recipient, don't run any other tests, even if feature delay-checks is turned on.
  - OK: override other checks for RCPT.
  - QUICK:OK: override other checks except for relay tests.
  - REJECT: reject recipient.
  - TEMP: tempfail recipient.
  - CONTINUE: continue with other checks.

  If the lookup returns neither REJECT nor RELAY then the usual anti-relay tests are performed.

  Note: SMAR and SMTPS have to interact here properly to avoid too much communication between them (and hence latency). The problem is that SMTPS may make some local decisions whether to allow relaying (based on local maps of some kind, e.g., regular expressions) and those checks are also part of the algorithm outlined above.

3.5.2.9 Anti-Spam Checks: API

The API provides functions for each SMTP command/phase similar to sendmail 8. For maximum flexibility, we specify it as asynchronous API as outlined in Section 3.1.1. In this case, two different function calls for the same session/transaction can not overlap (theoretically we could do that due to pipelining, but it probably doesn't give us anything). Therefore the session context (which may contain (a pointer to) the transaction context) should be enough to store the state between calls, i.e., it acts as a handle for the two corresponding functions.

check_connect(INOUT session-ctx, IN line, OUT state), check_connect_status(IN session-ctx, OUT state),

check_helo(INOUT session-ctx, IN line, OUT state), check_helo_status(IN session-ctx, IN line, OUT state),

check_ehlo(INOUT session-ctx, IN line, OUT state), check_ehlo_status(IN session-ctx, OUT state),

check_noop(INOUT session-ctx, IN line, OUT state), check_noop_status(IN session-ctx, OUT state),

check_rset(INOUT session-ctx, IN line, OUT state), check_rset_status(IN session-ctx, OUT state),

check_vrfy(INOUT session-ctx, IN line, OUT state), check_vrfy_status(IN session-ctx, OUT state),

check_expn(INOUT session-ctx, IN line, OUT state), check_expn_status(IN session-ctx, OUT state),

check_mail(INOUT session-ctx, IN line, OUT state), check_mail_status(IN session-ctx, OUT state),

check_rcpt(INOUT session-ctx, IN line, OUT state), check_rcpt_status(IN session-ctx, OUT state),

check_data(INOUT session-ctx, IN line, OUT state), check_data_status(IN session-ctx, OUT state),

check_header(INOUT session-ctx, IN buffer, OUT state), check_header_status(IN session-ctx, OUT state), called for each header line?

check_body(INOUT session-ctx, IN buffer, OUT state), check_body_status(IN session-ctx, OUT state), called for each body chunk?

It might also be possible to just pass the SMTP command/phase as parameter and hence minimize the amount of functions. However, then the callee most likely has to dispatch functions appropriately. sendmail X will most likely only provide functions for connect, mail, and rcpt, hence it might really be simpler to have just a single function. There must be some mechanism anyway to act upon configuration data (see Section 3.5.2.7).

ToDo: specify the behavior of SMTPS when function calls fail and the valid return codes and their effects. This should be consistent with the behavior specified in 3.5.2.7 and 3.5.2.8. See Section 3.5.4 for an attempt to specify this behavior.

Notice: if the anti-spam functions end up in a different module from the SMTP server, we certainly have to minimize the amount of data to transfer; it would be bad to transfer the entire session/transaction context all the time. Crazy idea: have a stub routine that only sends the changed data to the module. This might be ugly, but could be generalized. However, it would require that the stub routine recognizes changed data, i.e., it must store the old data and provide a fast method to access and compare it. Then we need an interface that allows to transfer only changed data, which could be done by using "named" fields. However, then also the other side needs to store the data and re-assemble it. Moreover, both caches need to be cleaned up, either explicitly (preferred, but would require extra API calls which may expose the implementation; at least we would need a call like close or discard) or by expiration (if an entry is expired too early, i.e., while in use, it would "simply" be added again). The overhead of keeping track and assemble/re-assemble data might outweigh the advantage of minimized data transfers.

3.5.2.10 Valid Recipient Checks

The SMTP server must offer a way to check for valid (local) users. This is usually done in two steps:

1. Identify local domains, i.e., check the domain part of an address.
2. Check the localpart against a list of valid users/aliases.

These steps could be combined into one, i.e., look up the complete address in one map. However, this may require many entries if a lot of "virtual" domains are used and valid users are the same in those. Moreover, a map of valid localparts might be the password file which does not contain domain parts, hence the check must be done in two steps.

3.5.2.11 Valid Sender Checks

The mail sender address must not just be syntactically valid but also replyable. The simplest way to determine this is to treat it as a recipient address and ask the address resolver to *route* (resolve) it. However, there are a few problems: the address resolver should not perform alias expansion because it would be a waste of time to check a long list of addresses (recipients). *Question*: if an address is found in the alias map, is that a sufficient indication that it is acceptable (i.e., routable)? Answer: most likely yes, it is at least very unusual to have an address in the alias map which is not valid (routable).

sendmail 8 performs a simple check for the sender address: it tries to determine whether the domain part of the address is resolvable in DNS. Even though this is a basic check, it is not sufficient as spammers simply create bogus DNS entries, e.g., pointing to 127.0.0.1. Hence it seems to be useful to perform an almost full address expansion as described above and then check the IP addresses against a list of *forbidden* values which are specified in some map. This approach has also a small problem: if the address is routed via mailertable then it could be directed to one of those *forbidden* values. There are two possible solutions to this problem:

1. Require an overriding entry for those address in the access map.
2. Do not check entries that have been found in mailertable against *forbidden* values.

Solution 2 seems to be useful because mailertable entries are most likely made by an administrator with a clue and hence are implicitly ok. However, implicit operations can cause problems[19], hence approach 1 might be better from a design standpoint[20], but less *user friendly*.

3.5.2.12 Map Lookups

Most map lookups for the SMTP servers are implemented by SMAR to avoid blocking calls in the state-threads application. SMTPS sends minimal information to SMAR, i.e., basically a key, a map to use for the lookup, and some flags, e.g., lookup the full address, the domain part, the address without details ("+detail"). SMAR performs the lookup(s) and send back the result(s). The result includes an error code and the RHS of the map (maybe already rewritten if requested?).

3.5.3 Policy Milter Interface Behavior

The normal operation of a policy milter is fairly easy to implement in SMTPS, it basically acts the same as querying QMGR or SMAR (access map) whether an SMTP command is allowed. A policy milter has also access to the mail body, which is not inspected by other modules, however, such functionality can be added if requested, e.g., some simple regular expression matching on headers and body lines. The behavior in case of errors can be specified for a milter while it is mostly fixed for the other modules, i.e., if QMGR fails to respond then the session is terminated (421). In general, a policy milter should be invoked via the API described in Section 3.5.2.9. However, the API is currently not completely specified, especially with respect to the behavior of SMTPS when various errors occur and when SMTP error codes are returned from the API.

For a milter several modes are allowed:

1. ignore any errors.
2. issue temporary errors for every command.
3. terminate the session.
4. sendmail 8 also allows to issue permanent errors for every command. However, it is unclear where this would be useful.

Based on the first three error modes, the following connection states are possible:

1. no milter configured: !USE-MILTER.
2. milter configured:
   (a) connection is established: USE-MILTER.
   (b) connection is not established:
      i. ignore any errors: ignore milter: !USE-MILTER (same as 1).

[19] "A bug is a feature that cannot be disabled".
[20] The current (2004-04-19) implementation accepts mailertable entries due to the way results are returned.

ii. issue temporary errors for every command: MILTER-TEMPFAIL.
iii. terminate the session: MILTER-SSD.

Overall this means there are three basically different states:

1. Do not use milter.
2. Use milter.
3. Milter is temporarily unavailable. In this SMTPS should try to reconnect to the milter periodically (some minimum delay between attempts should be specified as configuration option).

These states should be inherited by each SMTP session such that they can be changed locally. An error in one session does not necessarily mean that the entire connection is broken. Moreover, reconnect attempts should only be made for new sessions, making them during a session is not really useful because a milter usually needs all session related data, e.g., the connection information.

A milter registers with the server in which of the SMTP steps it is interested. Hence in each step the server checks those flags before it decides whether to send the information to the milter.

An SMTP server should react in basically the same way to results from a milter as it does for results from other anti-spam checks, e.g., the access map, as specified in 3.5.2.8. However, that specification is currently not complete because it does ignore lookup errors (see item 1 in Section 3.5.2.8). Section 3.5.4 tries to take errors into account and completely specify the behavior.

3.5.3.1 Interaction between Anti-Spam Checks and Milter

Question: should OK from access map disable policy milter checks (see also Section 2.10.2)? That would be "confusing" for a milter that wants that data, even though it would match the REJECT behavior because milter is not invoked in that case. However, this is not really a symmetric situation: REJECT means that the command is not accepted and hence in almost all cases it does not matter what a milter would do about it. for OK the milter might want to do something. However, that would be a contradiction (a misconfiguration), i.e., one part of the configuration explicitly accepts the command another rejects it. The right return value in this case is CONTINUE instead of OK. In order for the milter to keep track of the SMTP dialog (and hence the current state) it might be possible to send the information to the milter including the fact that the command has been accepted and hence cannot be rejected by the milter ("for your information only").

Note: the interaction between anti-spam checks and milter is even more complicated if feature delay-checks is turned on. In sendmail 8 this was a simple hack that only influenced the rulesets (by renaming the connect and mail check and calling them from the rcpt ruleset). In sendmail X this is solved in a different way: the checks are run at the appropriate stage in the SMTP dialogue, but the results may be delayed. Hence a connection might be rejected unless a recipient test overrides it ("spam friend"). *Question*: How should this interact with milter tests? Should they just run "normally" as the other tests and their results are delayed?

3.5.4 More Generic Return and Reply Code Behavior

This section tries to enhance the response model from 3.5.2.8 with the handling of lookup errors. If done right, it should describe the behavior for the functions specified in Section 3.5.2.9. There are various ways to deal with errors (as mentioned elsewhere (3.5.3):

1. ignore any errors (similar to the `-o` flags for maps in sm8, which causes a lot of problems because errors are silently ignored which must not be done here).
2. terminate the session (with a temporary error: 421 session/server shutting down).
3. issue temporary errors for every command.

It seems that only the first two modes are really useful: ignore errors or terminate the session on error. Issuing a temporary error for every command seems like a waste of time and resources; it could only be useful if the functions can "recover", i.e., if an error is really temporary and might not occur if the call is repeated, even without any actions by the caller. For a milter a communication error between SMTP server and libmilter will usually require a reconnect (and maybe a restart of the milter), hence closing a SMTP session is the simplest way to deal with the problem.

This can be made more complicated ("fine grained") by specifying one of these options for each function (step in the SMTP dialogue) individually instead of globally.

For now we consider only two error modes:

1. ignore errors (EM-IGNORE).
2. terminate the session with a temporary error (EM-SSD).

Here is a simple approach to the problem:

An anti-spam function has a return value (called "lookup result" elsewhere): SUCCESS, NOTFOUND, TEMPFAIL, PERMFAIL. If the result is TEMPFAIL or PERMFAIL (i.e., an error) then the system should have a default reply code (which could be different for each test). *Question*: can the function itself change those errors into a corresponding reply code? That is, if the return value is an error then return NOTFOUND if the error mode is EM-IGNORE and SUCCESS plus SSD (see 3.5.2.8) if the error mode is EM-SSD. This could be done by a generic wrapper, which might even implement different error modes per state as indicated above.

If this idea is feasible, then the algorithm described in Section 3.5.2.8 can be used without changes. *Note*: there is most likely some problem with this approach, but currently I cannot find (remember) it.

3.5.5 Internal Interfaces

The SMTP server is started and controlled by the MCP. Each server communicates with the queue manager and the address resolver (question: directly?).

3.5.5.1 Throttling

The SMTP servers must be able to reject connections if requested by QMGR to enable load control. This should be possible in various steps, i.e., by gradually reducing the number of allowed connections. In the worst case, the SMTP server must reject all incoming connections. This may happen if the system is overloaded, or simply if there is no free disk space to accept mail.

3.5.5.2 Removal of Entries from CDB

The original design required that the SMTP server removes a content entry from the CDB on request by the queue manager. This was done to avoid lockings overhead since the SMTP server is the only one which has write access to the CDB. However, in the current design the QMGR interacts directly with CDB (which should be implemented as library) to remove entries from it. This is done for several reasons:

- If there are multiple SMTP server processes (as currently intended) then we may run into locking issues nevertheless.
- If the items in CDB are independent from each other, then no locking problems occur, because the QMGR "knows" which entries are used for which purpose, i.e., it can unlink an entry from CDB after all its references have been removed (because all recipients have been taken care of). Only if a CDB implementation requires a centralized data structure to keep track of all CDB entries, then this design decision (removal of entries via QMGR) can cause problems. In that case, a CDB process might be necessary, or some other way to communicate the update to the right process that keeps track of the CDB entries.

3.6 Address Resolver

3.6.1 External Interfaces

The address resolver doesn't have an external interface. However, it can be configured in different ways for various situations. *Todo:* and these are?

Question: how to specify address resolving routines? Provide some basic functionality, allow hooks and replacements? How to describe? There are four different types of addresses: envelope/header combined with sender/recipient. It must be possible to treat those differently. For example, a pure routing MTA doesn't touch header addresses at all.

3.6.2 Valid Recipients

3.6.2.1 Valid Local Recipients

In Section 2.6.6.1 a configuration example was given which is slightly modified here:

```
local-addresses {
  domains = { list of domains };
  map { type=hash, name=aliases, flags={rfc2821, local-parts, detail}}
  map { type=passwd, name=/etc/passwd, flags={local-parts}}
}
```

See Section 3.2.3 how the list of valid domains can be specified, and Section 4.3.3 about a possible way to check whether a domain is in a list.

The current implementation (2004-07-27) uses the following hacks:

1. To specify the list of local domains mailertable entries are used with RHS `[127.0.0.2]` which indicates that LMTP over a local socket should be used.
2. To specify valid local users the alias map is used with RHS `local:`.

Item 1 could be modified a bit to use a different RHS, e.g., `local:` or `lmtp:`, to avoid using an IP address. The implementation must be modified accordingly (theoretically it might be possible to use a broadcast address but this is just another hack which may cause problems later). Item 2 should be extended by the more generic description given above in the configuration example.

Possible flags for map types are:

- rfc2821: map contains RFC 2821 addresses as keys (should this be with or without ¡¿?) For consistency it would be better to use angle brackets.
- local-parts: map contains the local part of addresses as keys.
- detail: map contains also +detail extensions.

3.6.3 Internal Interfaces

The address resolver must be able to access the usual system routines (esp. DNS) and different map types (Berkeley DB, NDBM, LDAP, ...). In addition to this, sendmail X.x should contain a generic map interface. This can either be done using a simple protocol via a socket or shared libraries (modules).

Question: should the address resolver receive a pre-parsed address (internal, tokenized format) or the string-representation (external format) of the address? This depends on where the address parser is called. It is non-trivial to pass the internal (tokenized) form via IPC because it is a tree with pointers in memory. So if we want to use that format, we need to put it into a contiguous memory region with the appropriate information (we could use pointers starting from the begin of the memory section, but is the packing/unpacking worth it?). Question: do we want the SMTP server to do some basic address (syntax) checking? That seems to be ok, but adds code to it. This may also depend on the decision who will do anti-spam checks, because the address must be parsed before that.

Basic functionality: return DA information for an address.

ar_init(IN options, OUT status): Initialize address resolver.

ar_da(IN address, IN ar_tid, OUT da_info, OUT status): start an AR task; optionally get the result immediately.

ar_da_get(IN ar_tid, OUT da_info, OUT status): get results for an AR task.

ar_da_cancel(IN ar_tid, OUT status): cancel an AR task.

Question: do we want to add a handle that is returned from the initialization function and that is passed to the various AR routines? We may want to have different ARs running, or we want to pass different options to the routines. If we use a handle, then we need also a terminate function that ends the usage of that handle.

Note: as explained in Section 3.4.5, the AR is not doing MX lookups. Those are either done in the QMGR or the DAs.

3.6.3.1 Data Structures

Resolved address format is a tuple consisting of

- DA: delivery agent. Question: can this be a list of delivery agents? If so, the DAs must use the same arguments, e.g., host to contact etc.

- host: host to contact; can also be other argument for the selected DA, e.g., path of socket, program path.

- address: address of recipient in the required form for the DA.

Question: should there be more elements like: localpart, extension, hostpart? It would be nice to make this flexible (based on DA?).

Notice: the result of an address resolver call can be a list of such tuples. This is more than MX records can do. By allowing a list of tuples we have maximum flexibility. For example, the address resolver could return lmtp,host0,user@domain; esmtp,host1:host2,user@domain. Question: what kind of complexity does this introduce? By allowing different DA the scheduling may become more complicated. Most MTAs allow at most several hosts, but only one address and DA. This flexibility may have serious impact on the QMGR, esp. the scheduling algorithms. Before we actually allow this, we have to evaluate the consequences (and the advantages).

Of course an address can also resolve to multiple addresses (alias expansion).

3.6.3.2 Error Behavior

Handling errors is an especially interesting problem in the address resolver. Even though it looks simple at first, the problem is complicated by the fact that multiple requests might have to be made. For example, getting the IP addresses of hosts to contact for an e-mail address involves (in general) MX and A record lookups, i.e., two phases where the second phase can consist of multiple requests. Handling an error in the first phase is simple: that error is passed back to the caller. Handling an error in the first phase is complicated: it depends on the kind of error and for which of the A records an error occurs. The main idea for the address resolution is to find a host to which the mail can be delivered. According to the requirements, this is one of the systems that is listed as "best MX host". Hence if a lookup of any A record for any of the best MX hosts succeeds it is sufficient to return that result; maybe with a flag indicated that there might be more data but the lookup for that failed in some way. That is, a partial result is better than insisting that all lookups return successfully.

3.6.3.3 Caching of Map Lookups

It might be useful to cache results of map lookups in the AR to avoid too many lookups, esp. for maps where lookups are *expensive*. This requires that similar to DNS a TTL (Time To Live) is specified to avoid caching data for too long. A TTL of 0 will disable caching. In general it would be better if the external maps provide their own caching since then the code in the AR will be simpler. Moreover, if several modules in sendmail X will use maps, then caching in each of the modules will just replicate data with the usual problems: more memory in use, possibility of inconsistencies, etc.

Note: this may be a future enhancement that is not part of sendmail X.0.

3.6.3.4 Threading Model

The AR is likely to call functions that may take a long time before they return a result. Those functions will include network activity or other tasks that have a high latency, e.g., disk I/O: searching for an entry in a large map. If we actually know all the places that can block, we could employ a similar threading model as intended for the QMGR, and hence even make the AR a part of the QMGR. However, it is unlikely that we are able to determine all possible blocking function calls in advance, especially if third-party software (or modules) are involved. Hence it is easier for programming to use a threading model in which the OS helps us to get around the latency, i.e., the OS schedules another thread if the current one is blocked. This will require more threads than are usually advisable for a high-performance program (e.g., two times the number of processors), since more threads can be blocked. However, it is probably still a bad idea to use one thread per task, since that will more or less relate to one thread per connection which definitely is bad for performance. The event thread library described in Section 3.20.5.1.1 can be configured to have reasonable number of worker threads and hence should be flexible enough to use for the implementation of the AR.

If all blocking calls actually are introduced by network activity, then state threads (see Section 3.20.3.1) would be an efficient way to implement the AR. However, that requires that all libraries which are used by the AR perform I/O only via the state threads interface which is not achievable because that would require to recompile many third-party libraries.

3.7 Initial Mail Submission

3.7.1 External Interfaces

The external interface of the mail submission program must be compatible (as much as possible) with sendmail 8 when used for this purpose. There will be less functionality overall, but the MSP must be sufficient to submit e-mails.

3.7.2 Internal Interfaces

Todo: to which other program does the MSP speak? See the architecture chapter.

3.8 Mail Delivery Agents

3.8.1 External Interfaces

Mail delivery agents must obey the system conventions about mailboxes. sendmail X.0 will support LDAs that speak LMTP and run as a daemon or can be started by MCP (on demand) such as `procmail` [pro].

3.8.2 Configuration

Mail delivery agents have different properties (compare mailer definitions in sendmail 8). These must be specified such that the QMGR can make proper use of the DA.

These properties are listed below (some of them are configuration options for the MCP, i.e., how to start DA). Notice: this list is currently almost verbatim taken from the sendmail 8 specification.

- Protocol to use, e.g., SMTP, LMTP, or invoking a program with command line arguments and feeding entire message to it via stdin (or socket). The latter probably will not be available in sendmail X.0.

- Various options for ESMTP: try always, try only if offered, never try it (even if offered).
- STARTTLS, AUTH, etc, capabilities.
- Maximum number of recipients per invocation and per session (if SMTP/LMTP; is there another protocol that could have multiple transactions per session?).
- The maximum number of message deliveries per session (connection). If a mailer requires a new invocation for each message, the limit can be simply set to one.
- DSN types (SMTP, "final" delivery).
- Rewriting set(s) for sender and recipient addresses (header/envelope).
- The end-of-line string.
- The maximum message size.
- The maximum line length in the message body.
- The maximum time to wait for the mailer.
- Look up the user part of the address in the alias database. Normally this is only set for local mailers.
- The user must have a valid account on this machine, i.e., mailbox lookup must succeed. If not, the mail is bounced.
- Look up ".forward" for recipient (based on username, user+detail, or even full address).
- Upper case should be preserved in user names or host names. Standards require preservation of case in the local part of addresses, except for those address for which your system accepts responsibility.
- Disables the loopback check. Normally when sendmail connects to a host via SMTP, it checks to make sure that this is not accidently the same host name as might happen if sendmail is misconfigured or if a long-haul network interface is set in loopback mode.
- Use the hidden dot algorithm.
- Do not look up MX records for hosts contacted via SMTP/LMTP.
- Do not attempt delivery on initial recipient of a message or on queue runs unless the queued message is explicitly selected during a queue run or an ETRN request.

MCP configuration:

- The pathname of the mailer, special name is used for SMTP; argument vector to pass to the mailer.
- The working directory for the mailer.
- The default user and group id to run as
- The nice(2) increment for the mailer

sendmail 8 options to fix mails by adding headers like Date, Full-Name, From, Message-Id, or Return-Path will not be in sendmail X(.0) as options for delivery, but should be in the MSA/MSP such that only valid mails are stored in the system. It's not the task of an MTA to repair broken mails.

Other sendmail 8 options that probably will not be in sendmail X.0:

- When an address that resolves to this mailer is verified (SMTP VRFY command), generate 250 responses instead of 252 responses. This will imply that the address is local.
- Escape lines beginning with "From " in the message with a '>' sign.
- This mailer wants UUCP-style "From" lines with the ugly "remote from host" on the end.
- Do not send null characters to this mailer.
- Do not include angle brackets around route-address syntax addresses. This is useful on mailers that are going to pass addresses to a shell that might interpret angle brackets as I/O redirection. However, it does not protect against other shell metacharacters. Therefore, passing addresses to a shell should not be considered secure.
- Use the route-addr style reverse-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC 821 section 3.1, many hosts do not process reverse-paths properly. Reverse-paths are officially discouraged by RFC 1123.
- The mailer wants a -f from flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).
- Do not insert a UNIX-style "From" line on the front of the message.
- Always run as the owner of the recipient mailbox. Normally sendmail runs as the sender for locally generated mail or as "daemon" (actually, the user specified in the u option) when delivering network mail. The normal behavior is required by most local mailers, which will not allow the envelope sender address to be set unless the mailer is running as daemon.
- If mail is received from a mailer with this flag set, any addresses in the header that do not have an at sign after being rewritten by ruleset three will have the "@domain" clause from the sender envelope address tacked on.
- Force a blank line on the end of a message. This is intended to work around some stupid versions of /bin/mail that require a blank line, but do not provide it themselves. It would not normally be used on network mail.
- Strip quote characters off of the address before calling the mailer.
- Do not include comments in addresses. This should only be used if you have to work around a remote mailer that gets confused by comments. This strips addresses of the form "Phrase <address>" or "address (Comment)" down to just "address".
- Normally, sendmail sends internally generated email (e.g., error messages) using the null return address as required by RFC 1123. However, some mailers do not accept a null return address. If necessary, error messages will be sent as from the MAILER-DAEMON.
- This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.
- Apply DialDelay (if set) to this mailer.

More sendmail 8 options that will not be in sendmail X.0:

- 8 bit, 7 bit (maybe later, sendmail X will do only "just send it")

3.8.3 Configuring Multiple DAs

See Section 2.8.2 about delivery classed and delivery instances for some basic terminology (which is not yet fixed and hence not used consistently here). There are several ways multiple delivery agents (instances) can be specified:

1. Different processes with different features.
2. A single process with *subclasses*.
3. Different processes with identical features.

Note: proposal 3 is probably required to make efficient use of multi-processor systems, i.e., if a DA is implemented using statethreads [SGI01] (see also Section 3.20.3.1) then the DA can only make use of a single processor, moreover, it can block on disk I/O. Additionally, if a DA can only perform a single delivery per incocation (process), then multiple processes can be used to improve throughput by providing concurrency.

Proposal 1 is required if the different features are too complicated to implement in a single process. For example, it does not seem to be useful to implement a DA that speaks SMTP and performs local delivery as a single process because the latter usually requires changing uids. However, it makes sense to implement a DA that can speak ESMTP, SMTP, and LMTP in a single process, maybe even using different ports as well as some other minor differences. Multiple processes are also useful if the DA process has certain restriction on the number of DA instances (threads) it can provide. For example, one DA may provide only 10 instances while another one may provide 100. However, such a restriction may also be achieved by other means, e.g., in the scheduler.

In some cases it might be useful to have a fairly generic DA which can be *instantiated* by external data, e.g., via mailertable entries that select a port number. Obviously one instantiation parameter is the destination IP address (it would not make sense to specify a class for each of them), but which other parameters should be flexible (specified externally to the configuration itself)? Possible parameters are for example port number and protocol (ESMTP, SMTP, LMTP). However, this data must be provided somehow to instantiate the values. Offering too much flexibility here means that these parameters must be configured elsewhere — outside the main configuration file — which introduces most likely an additional configuration syntax (e.g., in mailertable: `mailer:host:port`) which needs to be understood by the user and parsed by some program, such adding complexity in the configuration and the program. This is another place where it is necessary to carefully balance consistency (simplicity) and flexibility. In a first version, a less flexible and more consistent configuration is desired (which also should help to reduce implementation time).

3.8.3.1 Selecting and Naming Delivery Agents

The previous section showed several proposals how to specify multiple DAs. It is most likely that all of them will be implemented, because each of them is useful under different circumstances. This causes a problem: how to select (*name*) a DA? If multiple processes can implement the same DA behavior, then they either need the same name or some other way must be found to describe them such that they can be referenced (selected).

There will be most likely only two ways to select a DA: one default DA (e.g., `esmtp`) and the rest can be selected via mailertable.

A DA needs several *names*:

1. one by which the MCP can reference it: this is the name of a (delivery agent) section specified in the configuration; it is (completely?) local to MCP and used mostly for logging purposes. Note: this refers to a delivery agent instance (see Section 2.8.2, item 2).
2. one by which it can be selected in SMAR (mailertable): delivery class (see Section 2.8.2, item 1) identifier, e.g., `esmtp`, `lmtp`.
3. one by which the scheduler can select it; this is a delivery agent instance (see Section 2.8.2, item 2).

   The scheduler receives a delivery class identifier from smar. Based on that it has to find a delivery agent instance that provides the delivery class. There can be multiple DAs, and hence some selection algorithm must be implemented; see also below (item 4) about round robin selection etc.

Todo: need some terminology here to properly specify what is meant and unambiguously reference the various items.

Some random notes that need to be cleaned up:

1. MCP can start multiple processes for one "service". Problem: how can those processes be distinguished?
   (a) Pass them some counter as ID? That is not easy, how should it be done? Make the argument string a format string and use printf() to create the "real" argument, e.g.,

   ```
   args = "program -f sm.conf -i %d"
   snprintf(realargs, sizeof(realargs), args, id)
   ```

   Instead of using such a format it might be simpler to just have an option `pass_id = -i;` and then pass `-i id` as first argument (just insert it). MCP can give a unique id to each process that specifies this option.

   (b) The process id cannot easily be be used by smtpc itself as unique id because pid is 64bit on some OS, but the id is only a 32bit value (extending this to 64bit is just not worth the trouble; the protocol is dealing mostly with 32bit values). Moreover, a pid can be recycled fairly fast, but this is not desired with the ids used here: a process must cleanly terminate and be removed from all internal tables, before another process (of the same kind) can register with the same id.

   (c) Alternatively an id could be given by qmgr to smtpc during the initial exchange. How complicated it that? It's a bit ugly because the id should be used to identify the client (smtpc) during its communication with the server (qmgr). However, it might be possible to use the pid during the initial request and then send back an id for subsequent communication. Nevertheless, this requires some work on both sides (initial id and "real" id).

   Note: the SMTPC session/transaction id format expects an 8 bit id only. Possible solution: qmgr keeps an "internal" id for each smtpc which is used to generate the transaction/session ids but not anywhere else.

   min/max processes: who requests more processes? How do they stop? Idle timeout? Number of uses? On request?

2. It is also possible to have a DA start several processes of itself. Advantage: it can do the id generation itself. Disadvantage: this requires additional code to start, stop, and control processes, which is already available in MCP and should not be duplicated elsewhere. Even if it could be written in form of a library that can do the basic process control (restarting in case of some errors but not others, rate limiting, etc) it still requires some control features, e.g., starting processes on request.

3. Hierarchy:
   (a) multiple process specifications ("service").
   (b) one service with multiple instances (started by MCP).
   (c) multiple threads (tasks? thread is used here just as a "thread of execution", it does not refer to some implementation, e.g., POSIX or statethreads) in a single process.

   orthogonal to this: multiple classes within some level of the hierachy.

4. Define delivery classes (better name? *transport*?), i.e., their properties: protocol, port, etc, and then specify delivery agents which implement those mailers. A DA can implement multiple delivery classes. Other properties of a DA are the number of processes and threads it can provide. If multiple DAs provide the same delivery classes, then maybe add some scheduling policy (round robin, switch only to next if one DA is completely busy, etc.).

5. *Question*: is is necessary or at least useful to combine delivery classes into *delivery families* (better word?) such that all delivery classes in a delivery family are implemented by the same delivery agents? For example, all delivery agents that implement the ESMTP protocol can probably also provide any variant of it, e.g., different ports. *Question*: Which structures in QMGR are "per delivery class" (or per delivery family) instead of per delivery agent? For example, currently ready queues are per destination IPv4 address, they do not take the delivery class (or agent) into account. There is already some hack: selecting LMTP as protocol is done by a specific IPv4 address. It seems this must be generalized.

   Compare this with sendmail 8: there are different types of mailers specified by the `Path` field, e.g., those which use TCP/IP for communication with a program (`IPC`), delivery to files (`FILE`), and external programs for which the communication occurs via `stdin` and `stdout`. All IPC mailers can be treated as one delivery family.

6. *Question*: what is required for sendmail X.0? All delivery agents are currently of the same type, i.e., they all only implement ESMTP and LMTP. Hence there is currently no need to select a DA based on delivery classes. Therefore it is not necessary to list delivery classes in a delivery agent (as described in Section 2.8.1).

3.8.4 Internal Interfaces

Interface between QMGR and DA (see Section 3.4.5.1).

Question: what about outgoing address rewriting, header changes, output milters?

3.8.4.1 Status Information from DA to QMGR

Section 3.9.2.1 specifies the "special" case how the SMTP client returns status information for a delivery attemtp to the QMGR. That section is SMTP client specific, it must be extended (generalized) for other DA types. Important is whether the transaction was successful, failed temporarily or permanently, and whether per-recipient results are available. In case of failures: is this failure "sticky" or can another delivery attempt (with different sender/recipients) made immediately?

Question: which information does the QMGR really need and what is just informational (for display in the mail queue)? Compare `mci` in sendmail 8. The QMGR needs to know whether the error is permanent or temporary and whether it is "sticky", i.e., it will influence other connections to that host too. *Question*: anything else?

Note: if a DA cannot access the mail content (CDB) — which must be logged as an error — then the transaction is considered successful if the error code is `ENOENT` which may happen if someone removed the CDB entry. Other access errors are returned to the QMGR.

3.9 SMTP Client

3.9.1 External Interfaces

The external interface of the SMTP client is of course defined by (E)SMTP as specified in RFC 2821. In addition to that basic protocol, the sendmail X SMTP client implementation will support: RFC 974, RFC 1123, RFC 2045, RFC 1869, RFC 1652, RFC 1870, RFC 1891, RFC 1893, RFC 1894, RFC 2487, RFC 2554, RFC 2852, RFC 2920.

Todo: verify this list, add CHUNKING, maybe (future extension) RFC 1845 (SMTP Service Extension for Checkpoint/Restart).

We have similar choices here for the process model as for the SMTP server. However, we have to figure out which of those models is the best for the client, it might not be the same as for the server.

3.9.2 Internal Interfaces

The internal interface must be the same as for the other delivery agents, except maybe for minor variations.

3.9.2.1 Status Information from SMTP Client to QMGR

The result from the SMTP client for a delivery attempt can be subdivided into two categories: per session and per transaction.

For the session the result contains the session-id and a status, which is one of the following:

- connection failed, error code (time out, refused, ...)
- failed after connection (not 220, but 4/5xy greeting)
- failed after HELO/EHLO (SMTP reply)
- does not offer required features (STARTTLS, AUTH, DELIVERBY, ...)
- feature failed (STARTTLS, AUTH, ...)

For a transaction the result contains the transaction-id and a status, where status might be fairly complicated. The transaction may fail at any of the SMTP commands (MAIL, RCPT, DATA, final dot) because the server returns an SMTP error code, there can be an I/O error, or there can be an internal DA error.

Keeping track of the delivery status is done per recipient (not per transaction), hence an SMTP client must return status information that can be applied to each recipient of a transaction. This is done by returning a transaction status and a (possibly empty) list of individual recipient statuses, i.e., the status of each recipient which has not been accepted. The transaction status applies to all recipients for which no individual status is sent. Each recipient status is initialized to an temporary error. Only after the RCPT command has been sent to the server and a reply has been received the status is updated accordingly. A transaction is obviously only successful if the final dot was accepted by the server, in which case OK is the transaction status to send to QMGR. If a temporary error is received for any of the SMTP commands MAIL, DATA, final dot, or if there is an I/O error or an internal DA error, then the transaction status is a temporary error. If a permanent error is received for any of the SMTP commands MAIL, DATA, final dot, then the transaction status is a permanent error.

3.10 Milter

3.10.1 External Interfaces

The external interface for libmilter is described in the documentation that comes with the sendmail 8 source code distribution.

For the purpose of the first sendmail X release, the functionality will be restricted such that all functions that modify a mail are disabled; they should return a new error code, `ENOIMPL`.

The SMTP server needs some configuration support for a milter. For simplicity, there will be only a single milter. If multiple milters are supposed to be used, then some kind of "multiplexor" must be implemented on the libmilter side.

3.10.2 Internal Interfaces

The internal interface for milter will not use the sendmail 8 – libmilter protocol. That protocol is not able to deal with multiple SMTP sessions over a single connection. For the same reason, it is not possible to use ESMTP as protocol even though that would be a good choice because it is a standard and it offers extensibility. Using ESMTP requires one file descriptor for each (incoming) ESMTP connection to be able to distinguish between different ESMTP sessions. This causes problems on some platforms because `stdio` is limited to 256 file descriptors. Another restriction is imposed by `select(2)` which usually can handle up to 1024 file descriptors. While this restriction be circumvented by using `poll(2)`, it is obviously better to check only a single file descriptor for activity instead of hundreds or thousands.

For simplicity, the same basic protocol that is used between the SMTP servers and SMAR or QMGR will be used, which is based on RCB (see Section 3.16.11.1.1). The protocol is internal, and hence described in the implementation Section 4.9.

3.11 Miscellaneous Programs

3.11.1 Show Mail Queue

A program (`mailq`) should show the content of the mail queues. Currently there are three mail queues (2.4.1): incoming mail queue, deferred mail queue, and active queue. It should be possible to show the content of the first two, those are on persistent storage; the last queue is just a subset of the other two.

Question: how accurate must the output be? To achieve a completely accurate output all write (and delete) operations must be stopped in principle, i.e., write access to a queue must be locked. There might be some ways around this, e.g., while the program is reading the queue content and another thread (process) wants to update it those changes must be communicated to the reader, however, this is certainly not worth the programming effort. Locking a queue just to display its content does not seem to be reasonable: if a queue is large then accessing it could take a considerable amount of time during which other activities are blocked. In such a situation locking a queue just makes progress even slower, and hence adds to the problem (e.g., a manager asking an employee every few moments whether some work is done).

3.11.2 Force Queue Run

It might be useful to ask the queue manager to schedule certain entries for immediate ("as soon as possible") delivery. This will also be necessary for the implementation of ETRN.

3.11.3 Mailstats

Some statistics need to be available. At least similar to `mailstats` in sendmail 8. Maybe the rest is gathered from the logfile.

3.12 Security Hints

3.12.1 Owners and Permissions

This section describes one possible choice for user and group ids in the sendmail X system and the owners and permissions of files, directories, sockets etc.

The MCP is started by root. The configuration file can either be owned by root or a trusted user. Most other sendmail X programs are started by the MCP.

The following notation is used:

- Owner-Program: O-P
- Group-Program: G-P

where P is one of Q: QMGR, S: SMTPS, C: SMTPC (or other DAs).

There are the following files which must be accessed by the various programs:

- CDB: written by SMTPS, read by SMTPC, entries are removed by QMGR.

  Hence the directory must be writable by SMTPS and QMGR, accessible by SMTPC. The files must be writable by SMTPS and readable by SMTPC.

  Hence the directory is owned by SMTPS (O-S), has group QMGR (G-Q) and the following permissions 0771 (or 0731):

  ```
  d rwx rwx --x
  ```

  The files in the directory are owned by SMTPS (O-S) and have group SMTPC (G-C) with permissions 0640:

  ```
   rw- r-- ---
  ```

Todo: figure out how this works for non-Unix systems.

- IBDB: written and read by QMGR only.
- DEFEDB: written and read by QMGR only.

Moreover, there is one communication socket each between QMGR and other sendmail X programs, hence these sockets must be readable/writable by QMGR and that program. Considering that socket permissions are not sufficient to protect a socket in some OS, the socket must be in a directory with the proper permissions.

The sockets and directories are owned by QMGR and group-accessible to the corresponding program. Problem: either QMGR must belong to all those groups (to do a `chown(2)` to the correct group), or the directories must exist with the correct permissions and the group id must be inherited from the directory if a new socket is created in that group. The former can be considered a security problem since it violates the principle of least privileges. The latter may not work on all OS versions.

3.13 Databases and Caches for Envelopes, Contents, and Connections

This section deals in general with databases and caches as they are used in sendmail X.

General notice: us usual it is important to tune the amount of information stored/maintained such that the advantages gained from the information (faster access to required data) outweighs the disadvantages (more storage and effort to properly maintain the data). For example, data should not be replicated in different places just to allow simpler access. Such replication requires more storage (which in case of memory is precious) and the overhead for maintaining that data and keeping it consistent can easily outweigh the advantage of faster access.

3.13.1 DBs with Multiple Access Keys

Section 3.4.4 explains that some DBs need multiple access keys. This can be achieved by having one primary key and several secondary keys. Question: do we want to allow a varying number of keys, or do we want to write modules with one, two, three, and four keys? The latter is easier to achieve, but not as flexible. However, if the full flexibility is not needed (it most likely isn't), then it might not be worth to try to specify and implement it.

3.13.2 DBs with Non-unique Access Keys

In some cases an access key (which does not need to be the primary key, see 3.13.1) may not uniquely specify an element. For example, if a host name is used as key, then there might be multiple entries with the same key. In that case, we need to provide functions to return either a list of entries or to walk through the list. The latter is probably more appropriate, since usually the individual entries are interesting, not the list as a whole. For such "walk through" function we need a context pointer that is passed to the get-next-entry function to remember the last position.

Note: if non-unique access keys are used, then it might be still better to have a primary key which is unique. This should simplify some parts of the API and the implementation. For example, to remove an element from the DB it is required to uniquely identify it (unless the application wants to remove any element matching the non-unique key which might be only useful in a few cases). If we don't have a unique key, then the application needs to search the correct data first and pass a context to the remove function which is in turn used as unique identifier (e.g., simply the pointer to the entry, but the structure should be kept opaque for abstraction).

In some cases the application data itself can provide a list of elements with the same key such that the access method can simply point to one element (usually the head of the list). It is then up to the application to access the desired elements. However, it might be useful to provide a generic method for this, at least in the form of macros to maintain such a list. For example, if the element to which the access structure points is removed, the (pointer in the) access structure itself must be updated. Such a removal function should be provided to the application programmer to minimize coding effort (and errors).

3.13.3 Envelope Database Access Methods

First we need to figure out what kind of access methods we need. This applies to the incoming queue, the active queue, and all other queues. Then we can specify an API that gives us those access methods. Of course we have to make sure that we do not restrict ourselves if we later come up with another access method that is useful but can't be implemented given the current specification. We also have to make sure that the API can be implemented efficiently. Just because we would like some access methods doesn't mean that we should really implement them in case they lead to slow implementations.

Question: do we need a recipient id? This might be necessary if the same recipient is given twice. Even though we could try to remove duplicates, there might be different arguments for the recipient command. It might not be useful, but we have to be able to deal with it. Answer: we need a recipient id as key to access the data in the various EDBs.

There must be some way to search for entries in an EDB according to certain criterias, e.g., recipient host (delivery agent, destination host), delivery time, in general: scheduling information.

Collection of thoughts (just so they don't get lost, they need reconsideration and might be thrown away):

- Reverse MX: keep a map for MX to host (which of course obeys TTLs). If an MTA comes up check the map and access entries in the deferred queue via the destination host.
- IP to MX to host map for IPs which point to multiple MX/host names.

3.13.4 Incoming Envelope Database API

The incoming envelope database (INCEDB) is the place where the queue manager stores its incoming queue. It is stored in memory (restricted size cache) and backed up on disk. Question: do we need two APIs or can we get along with one? There are some functions that only apply to the backup, but that can be taken care of by a unified API. However, this unified API is just a layer on top of two APIs, i.e., its implementation will make use of the APIs specific to the RSC and the disk backup. For the queue manager, the latter two APIs should be invisible. For the implementation, it might be useful to describe them here.

Question: When do we need access to the backup? Answer:

1. Recovery (after shutdown or crash)
2. If the restricted size cache (RSC) overflows, i.e., some envelopes are only in the backup but not in the RSC. Question: Shouldn't we move in that case old entries into the deferred queue instead?

> That might simplify the API for the INCEDB, because then there would be only one time when we need to read data from it: at system startup. This access method can be slow without interfering with normal operation.

Decision: Use the disk-backup of the INCEDB purely for desaster recovery. So envelopes stay in the RSC if they can't be transferred to the ACTEDB as long as the RSC doesn't overflow. If it does we use the deferred EDB or we slow down mail reception.

Question: do we just store the data (addresses) in unchanged form in the INCEDB or do we store them in internal (expanded) form? In general we want the expanded form, but due to temporary problems during the RCPT stage external formats may be stored too. In that case the address must be expanded when read from the INCEDB before it is placed into the active queue. See also Section 3.13.6.

3.13.4.1 Incoming Envelope Database API: RSC

API proposal (this is certainly not finalized):

- iqdb_open(IN name, IN mode, IN size, OUT status, OUT iqdb-handle): open an incoming envelope database.
- iqdb_close(IN iqdb-handle, OUT status): close an incoming envelope database.
- iqdb_session_new(IN iqdb-handle, IN session-info, OUT status, OUT session-id): create a new session (client connects succesfully).
- iqdb_trans_new(IN iqdb-handle, IN sender-env-info, OUT status, OUT trans-id): create a new envelope (done on MAIL command).
- iqdb_rcpt_add(IN iqdb-handle, IN trans-id, IN rcpt-env-info, OUT status): add a new recipient (RCPT command).
- iqdb_rcpt_rm(IN iqdb-handle, IN trans-id, IN rcpt-id, IN rpct-status, OUT status): remove a recipient (mail has been taken care of). This includes a recipient status, e.g., successfully delivered, some kind of failure, and hence the recipient ended up in another queue. The caller will take care of DSN handling or putting the entry into another queue if necessary.
- iqdb_trans_rm(IN iqdb-handle, IN trans-id, OUT status): remove an envelope from the EDB (after all recipients have been taken care of). Should this be done implicitly when all recipients have been removed?
- iqdb_commit(IN iqdb-handle, IN trans-id, OUT status): commit envelope information to stable storage.
- iqdb_session_rm(IN iqdb-handle, IN session-id, OUT status): remove a session from the EDB.
- iqdb_trans_discard(IN iqdb-handle, IN trans-id, OUT status): discard envelope (connection has been aborted somehow: RSET, EHLO, connection error on lower level).
- iqdb_session_lookup(IN iqdb-handle, IN session-info, OUT status, OUT session-handle):
- iqdb_trans_lookup(IN iqdb-handle, IN sender-env-info, OUT status, OUT trans-handle):
- iqdb_rcpt_lookup(IN iqdb-handle, IN rcpt-env-info, OUT status, OUT rcpt-handle):

- iqdb_items_cache(IN iqdb-handle, OUT items, OUT status): return the number of items in the memory cache.
- iqdb_readprep(IN iqdb-handle, OUT cursor, OUT status): prepare to read through EDB.
- iqdb_getnext(IN cursor, OUT status, OUT record): Get next entry from EDB.
- iqdb_readclose(IN iqdb-handle, IN cursor, OUT status): stop reading through EDB.

If the RSC stores data of different types, it must be possible to distinguish between them. This is necessary for functions that "walk" through the RSC and perform operations on the data or just for generic functions. A different approach would be to use multiple RSCs each of which stores only data of a unique kind. However, that wastes memory (internal fragmentation), since it cannot be known in advance in which relations the data needs to be stored, e.g., there might be only a few sessions with lots of transactions, or there might be many sessions with one transaction each.

3.13.4.2 Incoming Envelope Database: Misc

The INCEDB stores envelope information only temporarily. The envelopes will usually be moved into the active queue. A reasonable implementation of the disk backup currently seems to be a logfile. This file shouldn't grow endlessly, so it must be rotated from time to time (based on size, time, etc). This is done by some cleanup task, which should be a low-priority thread. It reads the backup file and extracts that envelope data that hasn't been taken care of yet. These entries are consolidated into a new file and the files read will be removed or marked for reuse. The more often it runs the less memory it may need because then it doesn't need to read so many entries (hopefully). Question: should this be made explicit in the API or should this be a side effect of the commit operation, e.g., if logfile big enough, rotate it? It seems cleaner to have this done implicitly. But that might be complicated and it might slow down the QMGR in an unexpected moment. Making the operation explicit exposes the internal implementation and binds part of the queue manager to one particular implementation. This violates modularity and layering principles. However, we could get around this by making those functions (rotate disk backup) empty functions in other implementations. Decision: don't make the implementation visible at the API level. It's just cleaner. It can be some low-priority thread that is triggered by the commit operation.

Question: do we need to store session oriented data, e.g., AUTH and STARTTLS, in the INCEDB?

Question: how do we handle commits? Should that be an asynchronous function? That is, initiate-commit, and then get a notification later on? The notification might be via an event since the queue manager is mainly an event-driven program. This might make it simpler to perform synchronous (interactive) delivery, i.e., start delivery while the connection from the sender is still open and only confirm the final dot after delivery succeeded, which allows to not actually commit the content to stable storage. However, does this really help anything? There has to be a thread that performs the appropriate actions, so that thread would be busy anyway. However, most of the work is done by a delivery agent, so we don't need to block a thread for this. So we need another function: incedb_commit_done(IN incedb-handle, IN trans-id, IN commit_status, OUT status): trigger notification that envelope has been committed.

Question: should we use asynchronous operations (compare aio)? edb_commit(handle, trans-id, max-time): do this within max-time. edb_commit_status(handle, trans-id): check the status. How do we want to handle group-commits? We can't do commits for each entry by itself (see architecture section), so we either need to block on commit (threading: the context is blocked and when enough entries are there for committing or too much time passed, then the commits are actually performed and the contexts are ready for execution again), or we can use asynchronous operation. However, we don't want to poll, but we want to be notified. Does the API depend on the way we implement the queue manager? Since the

QMGR is the only one with (write) access to the EDB, it seems so. But we need a similar interface for the CDB (see Section 3.13.7.1). So we should come up with an API that is more or less independent of the callers processing model. On the other hand, it doesn't make sense to provide different functions to achieve the same, if we don't need it (compare blocking vs. non-blocking I/O). It only makes it harder for us to implement the API. *Todo:* figure out whether EDB and CDB can use similar APIs for commits and how much that depends on the callers processing model (thread per connection, worker threads, processes).

3.13.4.3 Incoming Envelope Database API: Disk Backup

The disk backup for the INCEDB should have its own API for the usual reasons (abstraction layer). API proposal (this is not finalized):

- ibdb_open(IN name, IN mode, IN size, OUT status, OUT ibdb-handle): open an incoming envelope database.
- ibdb_close(IN ibdb-handle, OUT status): close an incoming envelope database.
- ibdb_trans_add(IN ibdb-handle, IN sender-env-info, IN trans-id, OUT status): write envelope data.
- ibdb_trans_rm(IN ibdb-handle, IN trans-id, OUT status): remove an envelope from the EDB (after all recipients have been taken care of).
- ibdb_trans_discard(IN ibdb-handle, IN trans-id, OUT status): discard envelope.
- ibdb_rcpt_add(IN ibdb-handle, IN trans-id, IN rcpt-env-info, OUT status): add a new recipient (RCPT command).
- ibdb_rcpt_rm(IN ibdb-handle, IN trans-id, IN rcpt-id, IN rpct-status, OUT status): remove a recipient (mail has been taken care of).
- ibdb_commit(IN ibdb-handle, IN trans-id, OUT status): commit envelope information to stable storage.
- ibdb_readprep(IN ibdb-handle, OUT cursor, OUT status): prepare to read through EDB.
- ibdb_getnext(IN cursor, OUT status, OUT record): Get next entry from EDB.
- ibdb_readclose(IN ibdb-handle, IN cursor, OUT status): stop reading through EDB.
- ibdb_rcpt_app(IN ibdb-handle, IN trans-id, IN rcpt-env-info, INOUT ibdb-req-list, IN status): append a recipient change request.
- ibdb_ta_app(IN ibdb-handle, IN sender-env-info, IN trans-id, INOUT ibdb-req-list, IN status): append a transaction change request.
- ibdb_req_cancel(IN ibdb-handle, IN ibdb-req-list): cancel all the requests in the list.
- ibdb_wr_status(IN ibdb-handle, INOUT ibdb-req-list): perform the status updates as specified by the request list.

The last four functions deal with request lists. These lists contain status updates for transactions or recipients. Requests lists are used to support transaction based processing: all change requests are collected in a list and after all of the necessary operations have been performed, the requests are committed to disk. This is for example helpful in a loop that updates status information: instead of updating the status for each element one by one, the changes are collected in a list. If during the processing of the loop an error occurs which requires that all changes are undone, then the request list can simple be discarded (ibdb_req_cancel()). If the loop finished without errors, then the requests are committed to disk (ibdb_wr_status()). This also has the advantage of being able to implement group commits, which may result in better performance.

Question: do we need ibdb_trans_discard()? If the transaction data is only stored after the final dot, then we wouldn't need it. However, that might cause unnecessary delays, and in most cases mails are transmitted and accepted without rejected after the final dot (or transmission aborts due to timeouts etc). We could also merge ibdb_trans_rm() and ibdb_trans_discard() by adding a status parameter to a common function, e.g., ibdb_trans_end().

3.13.5 Active Envelope Database API

The active envelope database (ACTEDB) is the place where the queue manager stores its active queue, it is implemented as a restricted size cache. The active queue itself is not directly backed up by disk, but other queues on disk act as (indirect) backup.

Todo: clarify this API.

Routine	Arguments	Returns
actedb_open	name, size	status, actedb-handle
actedb_close	actedb-handle	status
actedb_env_add	actedb-handle, sender-env-info	status
actedb_env_rm	actedb-handle, trans-id	status
actedb_rcpt_add	actedb-handle, trans-id, rcpt-env-info	status
actedb_rcpt_status	actedb-handle, trans-id, rcpt, d-stat	status
actedb_rcpt_rm	actedb-handle, trans-id, rcpt-env-info	status
actedb_commit	actedb-handle, trans-id	status
actedb_discard	actedb-handle, trans-id	status

- actedb_open(IN name, IN size, OUT status, OUT actedb-handle): open active envelope database, specify size.
- actedb_close(IN actedb-handle, OUT status): close an envelope database.
- actedb_env_add(IN actedb-handle, IN sender-env-info, OUT status): add a new envelope (sender information contains also trans-id, maybe make it a separate parameter).
- actedb_env_rm(IN actedb-handle, IN trans-id, OUT status): remove an envelope from the ACTEDB (after all recipients have been taken care of).
- actedb_rcpt_add(IN actedb-handle, IN trans-id, IN int-rcpt, OUT status): add a new recipient.
- actedb_rcpt_status(IN actedb-handle, IN trans-id, IN rcpt, IN d-stat, OUT status) update recipient status and remove it from active queue. May also update the appropriate queue, i.e., incoming or deferred depending on where the entry came from and what the new status is. Another way to deal with that is to defer the update and perform a "group commit" just like for the incoming queue.
- actedb_commit(IN actedb-handle, IN trans-id, OUT status) commit envelope information to stable storage. Question: Shouldn't this be done separately for rcpt and mail? That is: actedb_mail_commit(IN actedb-handle, IN trans-id, OUT status) and actedb_rcpt_commit(IN actedb-handle, IN trans-id, OUT status). That may make it more complicated to have the data in sync. It

should really update all necessary data in one step (as much as this is possible, we probably won't implement real transaction based commits).

3.13.5.0.1 Delivery Agent DB API We need some support functions for the API specified in Section 3.4.5.1 to manipulate the Delivery Agent DB whose purpose and content has been explained in Section 3.4.10.12.

- dadb_open(IN da, IN da-descripton, OUT dadb-handle, OUT status)

  Open a DA DB.

- dadb_close(IN dadb-handle, OUT status)

  Close a DA DB.

- dadb_sess_open(IN dadb-handle, IN session, OUT dadb-entry, OUT status)

  Open a session and transaction. session contains the destination host to which to connect. The created dadb-entry contains newly created session and transaction handles.

- dadb_ta_open(IN dadb-handle, IN da-session-handle, OUT dadb-entry, OUT status)

  Open a transaction. The updated dadb-entry contains a newly created transaction handle.

- dadb_ta_close(IN da, IN da-trans-handle, OUT trans-status, OUT status)

  Close a transaction.

- dadb_session_close(IN da, IN da-session-handle, OUT status)

  Close a session.

3.13.6 Deferred Envelope Database API

The deferred envelope database (DEFEDB) is obviously the place where the queue manager stores the deferred queue.

Question: do we need session oriented data in the deferred EDB? Answer: shouldn't be necessary, all data required for delivery is in the envelopes (sender and recipients and ESMTP extensions). Note: this assumes that nobody is so wierd and uses session oriented data for routing. There might be people who want to do that; can those be satisfied by allowing additions to the data similar to "persistent macros" in sendmail 8?

We need several access methods for the EDB (from the QMGR for scheduling). Question: can we build indices for access "on-the-fly", i.e., can we specify a configuration option: 'use field X as index' without having code for each field? Answer: we probably need at least code per data type (int, string, ...). Question: is this sufficient? How about (DA, host) as index or something similarly complicated?

Todo: clarify this API.

Routine	Arguments	Returns
defedb_open	name, mode	status, defedb-handle
defedb_mail_add	defedb-handle, sender-env-info	status
defedb_env_rm	defedb-handle, trans-id	status
defedb_rcpt_add	defedb-handle, trans-id, rcpt-env-info	status
defedb_rcpt_status	defedb-handle, trans-id, rcpt, d-stat	status
defedb_rcpt_rm	defedb-handle, trans-id, rcpt-env-info	status
defedb_commit	defedb-handle, trans-id	status
defedb_readprep	defedb-handle	status, cursor
defedb_getnext	cursor	status, record
defedb_close	defedb-handle	status

- defedb_open(IN name, IN mode, OUT status, OUT defedb-handle): open an envelope database.
- defedb_close(IN defedb-handle, OUT status): close an envelope database.
- defedb_mail_add(IN defedb-handle, IN sender-env-info, OUT status): add an envelope (sender).
- defedb_env_rm(IN defedb-handle, IN trans-id, OUT status): remove an envelope from the EDB (after all recipients have been taken care of).
- defedb_rcpt_add(IN defedb-handle, IN trans-id, IN rcpt-env-info, IN d-stat, OUT status): add a new recipient (RCPT command), d-stat contains the delivery status.
- defedb_rcpt_status(IN defedb-handle, IN trans-id, IN rcpt, IN d-stat, OUT status) update the status of a recipient after a delivery attempt: successful/(temporary) failed delivery.
- defedb_rcpt_rm(IN defedb-handle, IN trans-id, IN rcpt-env-info, OUT status) remove a recipient from an envelope (recipient has been taken care of).
- defedb_commit(IN defedb-handle, IN trans-id, OUT status) commit envelope information to stable storage.
- defedb_readprep(IN defedb-handle, OUT cursor, OUT status) prepare to read through EDB.
- defedb_getnext(IN cursor, OUT status, OUT record) Get next entry from EDB.
- defedb_readclose(IN defedb-handle, OUT cursor, OUT status) stop reading through EDB.
- defedb_rcpt_get(IN defedb-handle, IN rcpt-spec, OUT status, OUT record) Get an entry from the EDB according to a recipient specification. Theoretically we could do an exhaustive search, but that would be bad for performance. It might be useful for mailq etc, but then a sequential read and a filter function should be sufficient.
- defedb_rcpt_app(IN defedb-handle, IN rcpt-spec, IN rcpt-status, OUT status): Append a recipient status change request to the internal list.
- defedb_ta_app(IN defedb-handle, IN sender-env-info, IN d-status, OUT status): Append a transaction status change request to the internal list.
- defedb_wr_status(IN defedb-handle): Write the internal list of status change requests to disk.

The last three functions deal with status change requests, similar to those for IBDB (see Section 3.13.4.3).

3.13.6.1 Deferred Envelope Database Implementation

Possible implementations:

1. A standard file-per-envelope implementation.
2. A database like Berkeley DB (BDB).
3. A "cyclical file system" version.

About option 3: The basic idea about a cyclical file system is to create a fixed set of files and reuse them. In this particular case we need files that represent the times at which queue entries should be tried again (theoretical maximum: Timeout.queuereturn, practical maximum: max-delay). Subdivide the maximum time into units, e.g., 1 minute, and use the files for the queue in round robin fashion. Each file represents the entries that are supposed to be tried at $now + n * time\_unit$. So a new entry (which represents a deferred mail/recipient) is placed into the file which represents its next-retry-time. If an entry is delayed again, it is appended to the appropriate file. Since the QMGR is supposed to take care of all entries that are to be tried *now*, the file will be afterwards "empty", i.e., no entry in the file is needed anymore.

Possible problems:

1. Since sender address and recipient addresses are treated separately, how do we relate those two? If we store the recipient addresses in those files, where do we store the sender addresses? Sender and recipient addresses are linked together via the transaction identifier. Hence we can use that identifier to access the sender address information for a recipient address, i.e., the transaction identifier is the key for the access, e.g., in a (very simple) database. However, we still need to design that file structure.

2. Granularity: 1 hour is too coarse, min-delay might be too small (what if it is 1 second?). Example: 5 days total, min-delay 1 minute: $60 * 24 * 5 = 7200$ files which can be easily taken care of by a two-level hashed directory structure, e.g., first level: hours ($5 * 25 = 120$), second level: minutes (60). Going down to 1 second as granularity requires 432000 files and probably a three level hash structure. Using 5 seconds granulariy requires 86400 files. Neither of these seems impossible to handle, but it seems to be unlikely to use such a fine granularity. Anything less than 10 seconds seems (currently) unreasonable, 1 minute seems ok.

3. Overflow: files cannot just denote $now + n$; what if the system is too busy and the *now* file cannot be taken care of within the granularity? Then we could overwrite a file. Hence we need at least some buffer space, i.e., we "artificially" increase the maximum time to max-delay plus some buffer space.

4. Variable length data: can those files be managed in the same way as the fixed length data? That is, can the files be reused after a certain time?

5. Items on hold (quarantined) or otherwise waiting for an event to be released should be treated differently. They are likely to be accessed by destination site or a match on a quarantine reason. Those entries could be held in a small set of files that is garbage collected from time to time similar to the incoming queue backup on disk. The admin may define different timeouts for these entries.

   Whenever an entry is added to the deferred queue a reason is specified (d-stat). The entry also contains scheduling information, see 3.4.10.6. This data can be used by the implementation to treat the entry accordingly.

6. There must be a (simple) way to update entries, i.e., the system must keep track of delivery attempts. When an entry is read from the (current) file and scheduled for delivery (or taken care of in some other way), then afterwards the new status must be noted somehow. This could be done by updating the file "in-place", see section 3.13.11.1. However, that might cause a lot of disk activity, esp. head movement, which should be avoided. A different approach, which is also consistent with this implementation, is to keep a logfile of entries that have been taken care of. When all entries in a file (block?) have been handled (use a bit map, maybe also a counter), the file can be removed or freed in some way, and therafter the logfile is removed/freed.

Note: Courier-MTA uses something remotely similar: it stores the queue files according to the next retry time, see courier/doc/queue.html.

> The mail queue consists of two directories: LSDIR/msgs and LSDIR/msgq. The control file is stored in LSDIR/msgs/nnnn/Ciiiii, and the data file is LSDIR/msgs/nnnn/Diiiii. "iiiiii" is the inode number of the control file. Since inode numbers are always unique, the inode is a convenient way to obtain unique identifiers for each message in the mail queue. "nnnn" is the inode number hashed, the size of the hash is set by the configure script (100 is the default, so this is usually just the last two digits of the inode number).
>
> One item of information that's stored in the control is the time of the next scheduled delivery attempt of this message, expressed as seconds since the epoch. There is also a hard link to the control file: LSDIR/msgq/xxxx/Ciiiiii.tttttt. "tttttt" is the next scheduled delivery time of this message. "xxxx" is the time divided by 10,000. 10,000 seconds is approximately two and a half hours. Each subdirectory is created on demand. Once all delivery attempts, scheduled for the time range that's represented by each subdirectory, have been made, the empty subdirectory is deleted. If a message needs to be re-queued for another delivery attempt, later, the next scheduled delivery time is written into the control file, and its hard link in LSDIR/msgq is renamed.
>
> This scheme comes into play when there is a large amount of mail backed up. When reading the mail queue, Courier doesn't need to read the contents of any directory that represents a time interval in the future. Also, Courier does not have to read the contents of all subdirectories that represent the current and previous time intervals, when it's falling behind and can't keep up with incoming mail. Courier does not cache the entire mail queue in memory. Courier needs to only cache the contents of the oldest one or two subdirectories, in order to begin working on the oldest messages in the mail queue.

About option 2: this may be simple because it requires only little implementation effort since the DB is already available. However, it isn't clear whether the DB is fast and stable enough. Considering the problems that other Sendmail projects encounter with Berkeley DB we have to be careful. Those problems are mainly related to locks help by several processes. In case a problem occurs and a process that holds a DB lock aborts without releasing the lock, all processes must be stopped and a recovery run must be started. In sendmail X we do not have multiple processes accessing the same EDB, only the QMGR has access to those. Hence the problem is minimized, we still need to make sure that in case of an abort the DB locks are properly released. However, we may even be able to use BDB without using its locks. If we lock the access to the DB inside the QMGR itself, then we avoid the DB corruption problem. Using locks inside the QMGR to sequentialize access to an EDB does not seem like a big problem. This would lead to some coarse-grain locking (obtain a mutex before accessing the DB), but that might be ok. It might even be possible to use BDB as cyclical filesystem, but that is something we have to investigate. The cyclical filesystem idea restricts the access methods to the recipient records to a purely sequential way, other access methods would be fairly expensive (require exhaustive search or at least other indices that may require many updates).

BDB provides a fairly easy way to use multiple indices to access the DB: the `DB->associate` method can link a secondary index to a primary and there can be multiple secondary indices (see [Sleb]: Berkeley DB: Secondary indices).

3.13.6.1.1 Deferred Envelope Database: Secondary Indices Instead of keeping secondary indices on disk (and hence increasing disk I/O which usually is a scarce resource), we may want to keep them in main memory and just restrict their size. Since they are just indices, i.e., they just contain a pointer (e.g., the main key for the DB), the size of each entry can be fairly small. For example, if the recipient identifier is used as main key then that's about 25 bytes (see Section 4.1.1). If then a secondary index is based on the 'next time to try' the overall size of the index is less than 32 bytes by entry (assuming a 4 byte time value). Hence even storing 100000 entries will only require about 3 MB of memory (only data; using an AVL tree will add about 20 bytes per entry on a 32 bit machine, hence resulting in a total of about 5 MB). If a system really has that many entries in the queue then the size of the secondary index seems fairly small compared to the amount of data it stores on disk (and the common memory sizes). Moreover, it should be possible to use restricted size caches for some indices and fill them up based on scanning the main queue. Coming back to our example of using 'next time to try' as secondary index, the data can be sorted (e.g., using AVL trees) and limiting the number of entries to some configurable value. Then we can simply remove entries that are too far away (their 'next time to try' is too far in the future for immediate consideration) when the cache is full and new entries must be added.

3.13.6.1.2 Deferred Envelope Database: Misc Note that not each transaction must be committed to disk immediately. It is not acceptable to lose mail, but it is acceptable (even though highly undesireable) to have mail delivered multiple times. Hence the information that a message has been accepted and not yet successfully delivered must always safely be stored on persistent storage. However, the fact that a mail has been delivered is not of that high importance, i.e., multiple of these informations can be committed in a group to minimize transaction costs. See also the sendmail 8 option CheckpointInterval and Section 3.13.11.

Note: transaction ids in old Berkeley DB versions (before 4.0.14 according to [Slea]) are 32 bit integers (of which only 31 bit are available) and hence may be exhausted rather soon. They are reset by running recovery, which hence might be necessary to do on a regular basis (at least on high volume systems).

3.13.7 Content Database

The content database stores the mail contents, not just the mail bodies (see 2.17).

Possible implementations:

1. A standard file-per-message implementation.
2. For implementations that expect high turnover of messages, a logfile-based CDB with scavenging garbage collection.
3. When almost all messages can be delivered or relayed right away, a delayed-commit implementation can conserve disk bandwidth.
4. Lastly, a content database that's integrated with a mail hosting message store can increase overall system throughput by a factor of two or more.
5. Wierd idea: use Berkeley DB as CDB.

Question: will the CDB be implemented as library or as a process by itself? In the former case there might be a problem to coordinate access to the CDB. If the CDB is accessed from different processes, then some form of locking may be required. This should be handled internally by the CDB implementation.

3.13.7.1 Content Database API

We need an API (access abstraction) that allows for different implementations. The abstraction must be good enough to allow for simple implementations (e.g., one file per mail content) as well as maybe complicated and certainly fast ones. It must be possible to use just memory storage if delivery modes allow for this (compare interactive delivery in sendmail 8), and it must be possible to use some remote storage on a distributed system. The API must neither exclude any imaginable implementation (as long as it makes some sense), nor must it keep us from doing extremely efficient implementations. It must accommodate (unexpected) changes in the underlying implementations during the lifetime of sendmail X.

Question: what exactly do we need here?

First we need to get a handle for a CDB. We may have to create a CDB before that but it seems better to do this outside the system with other programs, e.g., just like you have to set up queue(s) right now. It might be more flexible to create CDBs on demand (e.g., new directories to avoid "overflowing" the ones that exist), however, that doesn't seem to belong in the SMTP server daemon. If the CDB implementation needs it, it should do it itself whenever its operation requires it.

Question: what about different file systems for CDB? Maybe the user has a normal disk, a RAID, a memory file system, etc. So we need also some ideas how to use different file systems, which should be controllable by the user. Do we really want to make it that complicated? Can we hide this in the API and use "hints" in the open call? That should be good enough (for now); but it is sufficiently general?

We need to create a storage for the mail content per envelope. We should give it the transaction id and if existent the SIZE parameter. The library returns an id (content-id) and a status.

- cdb_start(IN name, IN mode, IN estimated-size, IN hints, OUT cdb-handle, OUT status): open a CDB. `mode` is read only or read/write (usually the latter). `hints` may contain information about the intended (estimated) usage, e.g., how many contents may be open at the same time.
- cdb_open(IN cdb-handle, IN trans-id, IN content-id, IN mode, IN size, IN hints, OUT status, OUT content-id): open a content entry. `mode`: read or write. `trans-id`: transaction id (can be NULL for read, then `content-id` must be supplied), `size` and `hints` can be supplied if `mode` is `write`.

  To make the interface flexible, there are two identifiers: transaction id as created by SMTPS and stored in the envelope data. If it proves to be useful or necessary, the CDB implementation can return its own identifier which will be used for further function calls (and will be stored in the envelope data too). It's not yet clear which data types to use, currently character strings are fairly likely. If the CDB doesn't want/need to generate its own ids, it can simply return the transaction id. If the CDB is realized as a library, the content-id may be an opaque pointer (or a small integer like a fd).
- cdb_write(IN cdb-handle, IN content-id, IN datap, IN len, OUT status, OUT written)
- cdb_read(IN cdb-handle, IN content-id, IN datap, IN len, OUT status, OUT read)

  Question: should read/write maintain the data pointer (offset) themselves (hidden behind content-id) or should this be passed back to the caller? If it is hidden, the calls might be optimized (assuming there is only one writer/reader at a time, which may not be true if we use interleaved delivery). The implementation must allow one writer and several readers concurrently.

- cdb_commit(IN cdb-handle, IN content-id, OUT status): make sure content is on persistent storage.
- cdb_close(IN cdb-handle, IN content-id, OUT status). Should close implicitly call commit?
- cdb_abort(IN cdb-handle, IN content-id, OUT status): obvious meaning.
- cdb_unlink(IN cdb-handle, IN content-id, OUT status): content isn't used anymore.
- cdb_stop(IN cdb-handle, OUT status): stop using this CDB.

Question: should all functions have cdb-handle as input? That allows to have multiple CDBs open at a time. It seems to be useful, because that makes the API more generic. However, the cdb-handle could be "hidden" in the content-id. For now, keep cdb-handle as parameter. If we don't use cdb-handle explicitly all the time, the interface is almost the same as for (file) I/O. Hence the question: do we want to use the same API? The I/O API call sm_io_setinfo() can be "abused" to implement calls like cdb_commit(), cdb_abort(), and cdb_unlink().

3.13.7.2 CDB: Group Commits

We have the same problem here as with the envelope DB: how to perform group commits? However, we probably can't use the solution (whatever it may be) from the QMGR in the SMTPS. The mechanism may depend on the process model of the respective module; it might be different in a program that uses worker threads from one that uses (preforked) processes or per connection threads. If we use per connection threads then we can simply use a blocking call: we can't do anything else (can we?) inbetween, we have to wait for the commit to give back the 250 (or an error). Moreover, most of the proposals how to implement the CDB do not even allow for group commits (unless the underlying filesystem provides this facility somehow). For example, if one file per mail content is used, then it is impossible in FFS to perform a group commit; each file has to be `fsync(2)`ed on its own.

3.13.7.3 Logging as CDB Implementation

Proposal 2 (see Section 3.13.7) basically tries to get around the problem of fast writes and group commits (synchronous meta-data operations). By using a log-structured filesystem (implemented by sendmail X or maybe just used on some OS that actually provide such a filesystem), writing contents to disk should be significantly faster (an order of magnitude) than using a conventional filesystem and one file per mail [SSB+95].

In the following, we take a look at some variations of this topic.

3.13.7.3.1 CDB: Log-Structured Filesystem For some references on log-structured filesystems and their performance relative to conventional filesystems see [RO92], [SBMS93], and [SSB+95]. Especially [SSB+95] indicates that proposal 2 (see Section 3.13.7) may not work as good as hoped. The main reason for this is garbage collection, which can decrease performance by up to 40%, which in turn causes LFS and FFS to have similar performance.

3.13.7.3.2 CDB: Cyclical Filesystem Question: can we use a cyclical filesystem also for the CDB? In addition to the problems mentioned in Section 3.13.6.1 for case 3 we also have a different garbage collection problem. If we store the mail content in files related to their arrival time, and do not reuse them until the maximum queue time is reached, then we have a huge fragmentation, but we would get garbage collection for free since we will simply reuse the storage after the timeout. How much was would this be? Let's assume we achieve a throughput of 10MB/s, that's 36GB/h, which is 864GB/day. Even though 100GB IDE disks can be bought for $200[21], that's a lot of waste. Moreso due to the number of disks and hence controllers required. Therefore this proposal seems too wierd, it requires too much hardware, most of which is barely used. Since most mail is delivered almost immediately, probably more than 90% of that space would be wasted, i.e., used only once during the reception and (almost immediate) delivery of a mail.

3.13.7.3.3 CDB: Logfile based Implementation It seems that garbage collections is the main problem for using log-structured filesystems as CDB for sendmail. However, maybe a property of the mail queue can help us to alleviate that problem. Garbage collection in conventional log-structured filesystems must be able to reclaim space in partially used segments (see [SSB+95] for the terminology). This requires compacting that space, e.g., by retrieving the "live" data and storing it in another segment and then releasing the old segment. Doing that requires of cause disk access and hence reduces the available I/O bandwidth. Therefore it should be avoided. Instead of reclaiming partially free segments only entirely free segments will be reclaimed, which doesn't require any disk access (except for storing the information on disk, i.e., the bitmap for free/used segments). This should be feasible without causing too much fragmentation since the files in the mail queue have a limited lifetime. Hence the general problem that a filesystem has to deal with doesn't (fully) apply here. It should be possible to reclaim most segments after a fairly short time. This approach might be something we can persue in a later version of sendmail X, for 9.0 it is definitely too much work. Even if we can get access to the source code of a log-structured filesystem (as offered by Margo Seltzer in [SSB+95], pg. 16), that code is in the kernel and hence would require a significant effort to rewrite. Moreover, it is not clear how much such a code would interfere with the (U)VM of the OS.

3.13.7.3.4 CDB: Berkeley DB based Implementation Proposal 5 looks like a strange idea. However, it may have some merits, i.e., it gives us group commits. If we use the transaction id concatenated with a block number as key, then the value would be the corresponding block. Group commits are possible due to the transaction logging facility of BDB. Whether this will give good performance (esp. for reading) remains to be seen (tested).

3.13.7.4 CDB: Meta Data Operations

General notice: currently it is claimed that meta-data operations in Unix file systems are slow and this (together with general I/O limits of hard disks) limits the throughput of an MTA. However, there are file systems that try to minimize the problem, e.g., softupdate (for *BSD), ext3, ReiserFS (Linux), and journalling file systems (many Unix versions). We may be able to take advantage of those file system enhancements to minimize the amount of work we have to do to implement a high performance MTA. Moreover, in high-performance systems disks with large NVRAM caches will be used, thus making disk access really fast.

3.13.7.5 CDB: File Reuse

Idea for simple CDB: keep queue files around and reuse them. That should minimize file creation/deletion overhead. However, does it really minimize meta-data operations? As it is shown in Section 5.2.1.2.1, some filesystems seem to become actually slower when reusing existing files.

[21] 2004: 200GB for $100

3.13.8 Connection Database (Cache)

The queue manager keeps track of all connections:

1. Outgoing connections that are currently open by the delivery agents.
2. Incoming connections that are currently open by the SMTP servers.
3. Past outgoing connections.
4. Past incoming connections.

The first two should be kept in memory, the last two can be stored on disk.

The open connections must be completely available in memory. The older connections can be limited to some maximum size, using something like LRU to throw expired connection out of the cache.

The connection caches should contain all information that might be relevant for the queue manager:

- Number of open connections
- Last time of connection
- Performance of connection:
  - bandwidth, latency (if easily available)
- Number of connection over some time period, e.g., 1, 2, 4, 8, ... minute(s).

For more details see 3.4.10.8 and 3.4.10.10.

3.13.9 Available Database Implementation

Do we want to look into other databases than Berkeley DB? How about:

- cdb (or better: the free alternatives, e.g., tinydb)? It is supposed to be very fast. However, it is optimized for reading, it should only seldom be rebuilt.
- DBZ from INN? (it is free, but it has very limited use).

3.13.10 Restricted Size Caches

Some data should be stored in memory but the amount of memory used should be limited. For example, the connection caches must be limited in size.

An RSC stores only pointers to data. We may need to create a copy of the data for the RSC for each new entry, in which case a entry_create() function should be passed to rsc_create().

- rsc_create(IN rpool_P rpool, IN size_t limit IN size_t htsize, IN int (*free_entry)(void *entry, void *context), IN void *context, OUT *rsc_t rsc, OUT status) create a new RSC with maximum size *limit* and a hash table size *htsize*.

  Returns a status and (if successful) a handle (otherwise NULL).

- rsc_lookup(IN rsc_t rsc, IN const char *key, IN unsigned keylen, OUT void *ptr, OUT status) lookup an entry. Returns a status and (if successful) a pointer to the entry (otherwise NULL).
- rsc_add(IN rsc_t rsc, IN bool delok, IN const char *key, IN unsigned keylen, IN void *elem, OUT void *ptr, OUT status) add (or update) an entry. Only if *delok* is set it is ok to delete an old entry if the RSC is full and a new entry must be added. Returns a status and (if successful) a pointer to the entry (otherwise NULL).
- rsc_rm(IN rsc_t rsc, IN const char *key, IN unsigned keylen, OUT status) remove (disable) an entry.
- rsc_usage(IN rsc_t rsc, OUT number) current usage. Returns how many entries are used. This is useful to figure out how full a cache (queue) is. This might return the total number of entries or a percentage (times 100 or scaled similarly).
- rsc_free(IN rsc_t rsc, OUT status) free (close) a RSC.

Question: how do we implement such caches? If the cache size is small, then we can create a fixed size array. We can then simply use a linear search for entries. Up to which size is this efficient enough?

If linear search becomes too slow, we need a better organization. We need two access methods: First by key, second by time. So we need to maintain two lookup structures for the array. The first one for the key can be a b-tree, the second one for the time can be a linked list or a b-tree.

We can use a ring buffer (fixed size linked list; fixed size array) for the time access. If the list is supposed to expire purely based on the time when an entry has been entered, then that organization is the simplest and fastet: when the buffer is full, just remove the tail (which is the oldest entry). If the expiration is based on LRU then a new entry can be made each time. The old one can be either removed (esp. if it is linked list), or marked as invalid (free). In the latter case the effective size would shrink which might not be intended but may be reasonable to get efficient operations.

One possible implementation can be taken from postfix: ctable.

3.13.10.1 Restricted Size Caches With Varying Size

For some structures it might be useful to vary the size of a cache depending on the load. Question: which ones are these? Is this really necessary? A possible implementation might be to link RSCs together, but that might be complicated if b-trees are used for lookups. Moreover, shrinking a cache is non-trivial, the question is whether it's worth to save some memory versus the computational overhead for shrinking.

3.13.11 Atomic Updates

For some operations atomic update are necessary or at least very useful. Such operations include updates of the delivery status of an email. In previous sendmail versions, `rename(2)` is used for atomic changes, i.e., to write a new qf file a tf fike is written and then `rename(tf, qf)` is used. BSD FFS provides this guarantee for `rename(const char *from, const char *to)`:

> `rename()` guarantees that if `to` already exists, an instance of `to` will always exist, even if the system should crash in the middle of the operation.

However, sendmail 8 doesn't update the delivery status of an envelope all the time by default:

> CheckpointInterval=N: Checkpoints the queue every N (default 10) addresses sent. If your system crashes during delivery to a large list, this prevents retransmission to any but the last N recipients.

sendmail X can behave similar, which means that atomic updates are not really necessary. However, it must made sure that there is no inconsistent information on disk, or that at least sendmail can determine the correct state out of the stored data.

3.13.11.1 Atomic Disk Writes

Question: are disk writes atomic? As long as they are within a block, are they guaranteed to leave the file in a consistent state? Can we figure out whether the write is within a block? If it spans multiple blocks, it may cause problems. Otherwise a write could be used for an atomic update, e.g., a reference counter or status update, without having to copy a file or create a new entry. If we can do "atomic one byte/block writes", we can update the information with an fseek(); write(new_status,1). Remark: `qmail-1.03/THOUGHTS` [Ber98] contains this paragraph:

> Queue reliability demands that single-byte writes be atomic. This is true for a fixed-block filesystem such as UFS, and for a logging filesystem such as LFS.

3.13.12 Data Structures for Queues on Disk and for Communication

If we store several queue file entries in a single file instead of each entry in an individual file (or some database), then the following layout instructions should be observed. Note: the same structure will be used for communication between sendmail X modules, see also Section 1.3.1 about this decision.

It might be useful to implement the queue files using a record structure with fixed length entries. This allows easy access to individual entries instead of sequentially reading through a file. To deal with variable length entries, continuation records can be used. Even if we don't use fixed size records, the entries should be structured as described below (tagged data including length specification). This allows for fast scanning and recognition without relying on parsing (looking for an end-of-record marker, e.g., CR).

The first record must have a version information and should contain a description of the content. It must have the length of the records in this file. Possible layout: record length (32 bit), version (32 bit), content description, maybe similar to a printf format string; see also Section 3.16.11.1.

Each record contains first the total length of the entry (in byte, must be multiple of the record length), then a type (32 bit), and then one or more subentries. The type should contain a continuation flag to denote whether this is the first or an continuation record. The last record has a length (less or) equal to the record length. It also has a termination subentry to recognize that the record is completely written to disk when it is read. Each of the subentries consists of its length (32 bit, must be multiple of 4 to be aligned), type (32 bit), and content. The length specifies only the length of the data, it doesn't include the type (or the length itself). We could save bytes if we do this the other way round: type first, then optionally length, whereby length can be omitted if the type says its a fixed size entry (32 bit as default). However, this probably makes en/decoding more complicated and is not worth the effort. Question: should each subentry be terminated by a zero character? If an entry doesn't fit into the record, a continuation record is used and the current record is padded, which is denoted by a special type. The termination subentry is identified by its type. Additionally, it might be useful to write the entire length of the record as value to minimize the problem of mistakenly identifying garbage on the disk as valid record. We won't use a checksum because the requires us to actually read through all the data before we can write it. Notice: variable length data should be aligned on 4 byte boundaries to allow things like accessing a (byte) buffer as an integer value. On 64bit systems it might be necessary to use 8 byte alignment.

Some open questions:

- Can we use `writev()` to write an entire record at once? If we use our strings (see 3.16.7) then we have the length of each entry, and we can directly compute the entire size (without using `strlen()` etc).
- How to break up records? Just fill `iovec` and check whether the record size is exceeded. If yes pad it and use the next record. What if an entry is longer than the record length? In that case we need to split it across records.
- Should numbers be printed in readable format or just as bit-pattern? In the latter case we either have to use `htonX()` or at least denote the endianness in the "header", i.e., the first record.

Note: see also xdr(3).

Note: this might be not needed if we go with Berkeley DB for the envelope DB. However, the "queue" access type for Berkeley DB works only with fixed sized records, so there may be a use for the following considerations.

3.13.12.1 Size Estimates

Let's do some size estimates for INCEDB, i.e., the disk back (see Section 3.13.4.3 and 3.4.10.4). The smallest size should be 4 bytes (32 bit word); for each entry a description (type) and length field is used additionally (see above).

entry	length (bytes)
transaction-id	20, up to 32 or even 64
start-time	8
cdb-id	up to 64
n-rcpts	8
sender-spec	up to 256 (RFC 2821) plus ESMTP extensions
size	8
bodytype	4
envid	up to 100 (RFC 1891)
ret	4
auth	no length restriction? (RFC 2554, refers to 1891)
by	10 for time, 1 (4?) for mode/trace

entry	length (bytes)
transaction-id	20, up to 32 or even 64
rcpt-spec	up to 256 (RFC 2821) plus ESMTP extensions
	maybe a unique id (per session/transaction?)
notify	4
orcpt	up to 500 (RFC 1891)

If we use a default size of 512 bytes, then we would get 100 KB/s if the MTA can deal with 1000 msgs/s (for one recipient per message, if it would be two – which is more than the average – it still would be only 150 KB/s). This is fairly small and should be easily achievable. Moreover, it doesn't amount to much data, e.g., for one day (without cleanup) it would be 12 GB (for very large traffic, i.e., 86 million messages).

3.13.13 Misc

Some questions and ideas; this part needs to be cleaned up.

Generic idea (see the appropriate sections earlier on which may have more detailled information): EDB: maybe use some of those as files (just append) and switch to a new one at regular intervals (time/size limits). The old one will be cleaned up then: all entries for recipients that have been delivered will be removed (logged), new file will be created. This has to be flexible to deal with "overflow", i.e., many of these files might be used, not a fixed set.

In some cases it might be useful to store domain names in reverse order in a DB (type btree). Then a prefix based lookup can (sequentially) return all subdomains too. Whether `host.domain.tld` should be stored as `dlt.niamod.tsoh` or `tld.domain.host` remains to be seen (as well as whether this idea makes much sense at all).

3.14 Maps

As explained in Section 2.12 maps are used to lookup keys and possibly replace the keys with the matching RHS. The required functionality exceeds that of default lookups (exact match) which can be provided by performing multiple lookups in which parts of the key are omitted or replaced by wildcard patterns (see 3.14.2). Accordingly replacement must provide a way to put the parts of a key that have been omitted or replaced into the RHS; this is explained in Section 3.14.3.

3.14.1 Results of Map Lookups

A map lookup has two results[22]:

1. the result of the lookup attempt, i.e.,
   - FOUND: the key was found in the map,
   - NOTFOUND: the key was not found in the map,
   - TEMPFAIL: the map lookup failed temporarily, another try later on may succeed (timeout, resource problem, etc),
   - PERMFAIL: the map lookup failed permanently.
2. iff the result is FOUND then a value (RHS, data) is returned.

3.14.2 Lookup Functionality

Section 2.12 explains the problem of checking subdomains (pseudo wildcard matches). The solution chosen[23] is a leading dot to specify "only subdomains" (case 2.12 in Section 2.12).

Below are the cases to consider. Note: in the description of the algorithms some parts are is omitted:

[22] *Question*: what is a useful terminology to distinguish those two *results*?

[23] At least for now, there might be a configuration option that determines whether a leading dot is required.

- If a lookup succeeds, stop and return the result.
- A flag indicates whether also just "`tag:`" should be looked up (to retrieve a default value).

1. IP addresses: Input: tag and IP address (printable format). Algorithm: lookup "`tag:IPaddress`", remove the least significant part of the IP address (i.e., the rightmost number including the dot) and try again: lookup "`tag:IPnet`"; repeat until all components are removed.
2. Hostnames/Domains: Input: tag and hostname. Algorithm: lookup "`tag:hostname`", if that failed: remove the first component from hostname and lookup "`tag:.rest`" (i.e., with leading dot); if there is something left repeat previous step. *Question*: should the last lookup be: "`tag:.`" or "`tag:`"?
3. E-mail addresses: Input: tag, user, detail, hostname. Algorithm:
   (a) Repeat the following lookups for each subdomain of hostname (see case 2: for subdomains a leading dot is used):
      i. "`tag:user+detail@hostname`" if detail is not `NULL`
      ii. "`tag:user++@hostname`" if detail is not `NULL` and not empty,
      iii. "`tag:user+*@hostname`" if detail is not `NULL`,
      iv. "`tag:user@hostname`"
      v. "`tag:hostname`"
   (b) If nothing has been found and hostname isn't `NULL` then try localpart only:
      i. "`tag:user+detail`" if detail is not `NULL`
      ii. "`tag:user++`" if detail is not `NULL` and not empty,
      iii. "`tag:user+*`" if detail is not `NULL`,
      iv. "`tag:user`"

   Notes:
   (a) If a component is `NULL` it will be omitted (unless the entire lookup depends on it).
   (b) A flag decides whether `@` is appended to the local part or prepended to the domain part. This is important for those lookups in which case one part can be `NULL`, for example, for aliases `@` would be prepended to the domain part, while for other lookups it would be appended to the local part. See also Section 3.14.2.1.
   (c) *Question*: should case 3(a)v be done inside the main loop or should that be done after all the cases 3(a)i to 3(a)iv have been tried for all subdomains? It might even be tried only after 3b.

3.14.2.1 Placing the '@' sign

In sendmail 8 there are various places where (parts of) e-mail addresses are looked up: aliases, virtusertable, access map. Unfortunately, all of these use different patterns which are inconsistent and hence should be avoided:

- aliases: `user`, `user@host`
- virtusertable: `user@host`, `@host`
- access map: `user@host`, `user@`, `host`

Question: should this be made consistent? If yes, which format should be chosen? By majority, it would be: `user@host`, `user`, `@host`. A compromise could be to always use the '@' sign to distinguish an e-mail address (or a part of it) from a hostname or something else: `user@host`, `user@`, `@host`.

3.14.3 Replacing Patterns in RHS

The RHS of a map entry can contain references to parts of the LHS which have been omitted or replaced into the RHS. In sm8 this is done via `%digit` where the arguments corresponding to the digits are provided in the map call in a rule. Since sendmail X does not provide rulesets, a fixed assignment must be provided. A possible assignment for case 3 (see previous section) is:

digit	refers to
0	entire LHS
1	user name
2	detail
3	+detail
4	omitted subdomain?

3.15 Error Handling

3.15.1 Returning Errors

Functions can return errors in different way:

- the return value is "invalid", i.e., not in the domain of valid results, e.g., NULL for functions that return a pointer, or less than zero for functions that return only positive integers as valid results. In that case an additional, detailed error code must be returned. Using a global variable (like `errno`) is unacceptable.
- all function results can be valid, hence the error code is given as a result parameter and must be explicitly checked.
- the return value is the error code, other results are output parameters (if necessary). This is used by the Pthreads API.

We will not use one general error return mechanism just for sake of uniformity. It is better to write functions in a "natural" way and then choose the appropriate error return method.

Question: is timeout an error? Usually yes, so maybe it should be coded as one.

Question: Do we want to use an error stack like OpenSSL does? This would allow us to give more information about problems. For example, take a look at `safefile()`: it only tells us that there is a problem and what kind of problem, but it doesn't exactly say where. That is: `Group writable directory` can refer to any directory in a path. It would be nice to know which directory causes the problem.

OpenSSL uses a fixed size (ring) buffer to store errors (see `openssl/include/err.h`), which is thread specific.

3.15.2 Error Classification

A detailed error status consists of several fields (the status, i.e., all of its fields, must be zero if no error occurred):

1. error type: temporary, permanent, and internal; the latter is a programming error, e.g., an interface violation. Warnings can be treated as a special form of errors too. If a resource limit is exceeded, it might be useful to include whether the problem is due to a soft or hard limit (`setrlimit(2)`), a compile or run time limit, or an OS limit. This can probably be encoded in 4 to 8 bits.

2. error code: a value similar to `errno` that exactly gives the error. This can probably be encoded in 16 bits.

3. subsystem: which part of the program (libraries, module) generated this error? This should be encoded in the remaining 8 to 12 bits to give a 32 bit error status.

As explained in 3.15.1 some functions return positive integers as valid results, hence negative values can be used to indicate errors. To facilitate this, the error codes will always be negative, i.e., the topmost bit is set.

So overall the error status is encoded in a signed 32 bit integer (of course `typedef` will be used to hide this implementation detail). Macros will be used to hide the partitioning of the fields to allow for changes if necessary.

In some cases a numeric error status is not sufficient, an additional description (string) is required or at least useful. This can be made available as a result parameter, however, see also the next section how errors can be returned.

3.15.3 Converting Error Codes into Textual Descriptions

If modules are used, it is hard to have a single function that converts an error code into a textual description like `strerror(3)` does as Section 3.17 explains. The subsystem field in the error value can be used to call the appropriate conversion function.

If we want to be extremely flexible, then a module can register itself with a global error handler. It submits its own error conversion function and it receives a subsystem code. This would make the subsystem code allocation dynamic and will avoid conflicts that otherwise may arise. Question: is this too complex?

3.15.4 Misc

Minor nit about tags in structures for assertion that the submitted data is of the right type. 8.12 uses strings, a nice integer value is sufficient, e.g., 0xdeadbeef, 0xbc2001. Then the check is just an integer comparison instead of a strcmp(). Currently it's just a pointer comparison, so it's probably not much difference. Question: is this guaranteed to work? Someone reported problems on AIX 4.3, it crashes in the assertion module. The reason for that is a bug in some AIX C compiler that does not guarantee the uniqueness of a C const char pointer. Bind 9 uses a nice trick defining a magic value as the "concatenation" of four characters (in a 32 bit word). Even though this restricts the range of magic values, at least it helps debugging since it could be displayed as string.

3.16 Libraries

sendmail X should make use of many libraries such that the source code of the main programs is only relatively small and easy to follow. Most complexity should be hidden in libraries, such as different I/O handling with timeouts and for different OS as well as different actual lower layers (e.g., IPv4 vs

IPv6). The more modules are implemented via libraries, the easier it should be to change the underlying implementations to accomodate different requirements and to actually reuse the code in other parts of sendmail X as well as other projects.

3.16.1 Naming Conventions

3.16.1.1 Naming Conventions for Types

_T	type
_E	enum
_F	function
_P	pointer
_S	structure
_U	union

`_T` is default, the other should be only used if necessary.

3.16.1.2 Naming Conventions for Functions

`_new()` or `_create()`	create a new object
`_destroy()`	destroy object (other name?)
`_open()`	open an object for use, this may perform an implicit `new()`
`_close()`	close an object, can't be used unless reopened, this does not destroy the object
`_add()`	add an entry to an object
`_rm()`	remove an entry from an object
`_alloc()`	allocate an entry and add it to an object
`_free()`	free an entry and remove it from an object
`_lookup()`	lookup an entry, return the data value
`_locate()`	lookup an entry, return the entry

There are many alternative names used for `lookup`: `search`, `find`, and `get`. The latter is used by BBD and even though it implies that a value is returned, it is a bit overloaded by the I/O function with that name, which return the next element, without implying that a specific entry is requested.

3.16.1.3 Naming Conventions for Variables

`X_ctx`	context for object X
`X_next`	pointer to next element for X
`X_nxt`	pointer to next element for X (if X is a long name)
`_state`	state (if the object goes through various states/stages)
`_status`	status: what happened to the object, e.g., SMTP reply code
`_flags`	(bit mask of) flags that can be set/cleared/checked more or less independent of each other

The distinction between state and status is not easy. Here's an example: A SMTP server transaction goes through various stages (states): none, got SMTP MAIL/RCPT/DATA command, got final dot. The status should be whether some command was accepted, so in this case the two are strongly interrelated. It probably depends on the object whether the distinction can be made at all and whether it's useful.

For example, the status might be stored in the substructures, e.g., the mail-structure has the response to the SMTP MAIL command.

3.16.1.4 Naming Conventions for Structures

As shown in Section 3.16.1.1 a structure definition in general looks like this:

```
typedef struct abc_S  abc_T, *abc_P;
struct abc_S
{
  type1    abc_ctx;
  type2    abc_status;
  type3    abc_link;
};
```

`X_link`	link for list X
`X_lnk`	link for list XYZ (if X is a long name)
`X_l`	link for list XYZ (if X is a really long name)

Note: `link` is used for structure elements instead of `next` which is used for variables. This makes it easier to distinguish variables and structure elements.

3.16.1.5 Naming Conventions for Macros

The name of macros should indicate whether they can cause any side effects, i.e., whether they use their arguments more than once and whether they can change the control flow. For the former, the macro should be written in all upper case. For the latter, the macro should indicate how it may change the control flow as depicted in the next table.

`_B`	break
`_C`	continue
`_G`	goto
`_E`	jump to an error label
`_R`	return
`_T`	terminate (exit)
`_X`	some of the above

Macros that include assert statements don't need to be of any special form, even though `_A` could be used; however, many functions use assert/require too without indication of doing that in their name.

3.16.2 Include Files

All include files in sendmail program code will refer to sendmail X specific include files. Those sendmail X specific files in turn will refer to the correct, OS dependent include files. This way the main program modules are not cluttered with preprocessor conditionals, e.g., `#if, #ifdef`.

3.16.3 I/O

sendmail X will use its own I/O layer, which might be based on the libsm I/O of sendmail 8.12. However, it must be enhanced to provide an OS independent layer such that sendmail X doesn't have to distinguish in the main code between those OSs. Moreover, it should be a stripped-down version containing only functions that are really needed. For example, it should have record oriented I/O functions (record for SMTP: a line ending in CRLF), and functions that can deal with continuations lines (for headers).

3.16.3.1 Required I/O Functionality

How should the I/O layer work?

It should buffer reads and writes in the following manner:

- read only: read full buffer, satisfy requests from there.
- write only: put strings into buffer, write buffer if full.
- read/write: two cases:
  - single buffered, e.g., a file. The problem here are "direction" changes, i.e., from reading to writing and vice versa.
    - reading to writing: discard read buffer.
    - writing to reading: flush the buffer.

    These cases become complicated in stdio since they have to deal with ungetc-buffers. If we restrict ungetc to push at most one character back and to do this only if previously a character has been read, then we can simplify this code significantly.
  - double buffered, e.g., a network connection. Reads and writes are independent (two buffers required).

stdio is also fairly complicated since it provides different buffering modes.

Requirements of SMTPS for data:

How to read the data with the least effort? We cannot just read a buffer and write it entirely to disk (give it to the CDB) because we need to recognized the trailing dot. The smtpsdata() routine should have access to the underlying I/O buffer. We could either be ugly and access the buffer directly (which violates basic software design principly, e.g., abstraction), or we can provide a function to do that. In principle, sm_getc() is that function, but it can trigger a read, which we don't want. If the buffer is empty, we want to know it, write the entire buffer somewhere, and then trigger a read. We could add a new macro: sm_getb() that returns SM_EOB if the buffer is empty. Note: to do this properly (accessing the buffer almost directly) we should provide some form of locking. Currently it's up to the application to "ensure" that the buffer isn't modified, i.e., the file must not be accessed in any other way inbetween.

3.16.3.2 I/O Buffer

The I/O buffer currently consists of the following elements:

int	size	total size
int	r	left to read
int	w	left to write
int	fd	fileno, if Unix fd, else -1
sm_f_flags_T	flags	flags, see below
uchar	*base	start of buffer
uchar	*p	pointer into buffer

This implements a simple buffer. The buffer begins at base and its size is size. The current usage of the buffer is encoded in flags. The interesting flags for the use as buffer are:

RD	currently reading
WR	currently writing

RD and WR are never simultaneously asserted. RW open for reading and writing, i.e., we can switch from one mode to the other.

The following always hold:

flags&RD $\Rightarrow$ w = 0

flags&WR $\Rightarrow$ r = 0

This ensures that the getc and putc macros (or inline functions) never try to write or read from a file that is in 'read' or 'write' mode. (Moreover, they can, and do, automatically switch from read mode to write mode, and back, on "r+" and "w+" files.)

r/w denote the number of bytes left to read/write. p is the read/write pointer into the buffer, i.e., it points to the location in the buffer where to read/write the next byte.

```
|<-        size         ->|
|----------------|--------|
^                ^        ^

base             p    base + size
```

RD $\Rightarrow p + r \leq base + size$

WR $\Rightarrow p + w \leq base + size$

The buffer acts as very simple queue between the producer and the consumer. "Simple" means:

- RD: the buffer is always completely read and then filled from the base (but maybe not completely filled since there might not be enough data).
- WR: the buffer is written and then completely flushed such that is empty again.

"Real" queues would have different pointers to write/read for the producer and consumer such that the queue is entirely utilized. However, that causes problems at "wrap-around", esp. if more than just one item (byte) should be read/written: it must be done piecewise (or at least special care must be taken for the wrap-around cases). We can do something like that because $p + r$ is our pointer to write more data into the buffer.

3.16.4 Event Driven Loop

The main part of most sendmail X programs is an event driven loop.

For example, the SMTP server reads data from the network and performs appropriate actions. The context for each server thread contains a function that is called whenever an I/O event occurs. However, if we use non-blocking I/O (which we certainly do), we shouldn't call a SMTP server function (e.g., smtps_mail_from) on each I/O event for that thread, but only if complete input data for that middle-level function is available. As stated before (see 3.16.3) the I/O layer should be able to assemble the input data first. So the thread has an appropriate I/O state which is used by the I/O layer to use the right function to get the input data. During the SMTP dialogue this would be an CRLF terminated line, if CHUNKING is active it would be an entire chunk.

Event driven programming can be fairly complicated, esp. if events can occure somewhere "deep down" in a function call sequence. We have to check which programs are easy to write in an event based model, and how we can deal with others (appropriate structuring or using a different model, e.g., threads per connection/task).

If a blocking call has to be made inside a function, it seems that the function must be split into two parts such that the blocking call is the divider, i.e., the function ends with a call that initiates the blocking function. The blocking function must be wrapped into an asynchronous call, i.e., an "initiate" call and a "get result" call. The function must trigger an event that is checked by the main loop. Such a wrapper can be implemented by having (thread-per-functionality) queue(s) into which the initiate calls are queued. A certain number of threads a devoted to such queue(s) and take care of the work items. When they are done they trigger an (I/O?) event and the main event loop can start the function that receives that result and continues processing. This kind of programming can be become ugly (see above).

3.16.5 Resource Pools

sendmail X will reuse the rpool abstraction available in libsm (with the exception of not using exceptions). Moreover, there should be an `rpool_free_block()` function even though it may actually do nothing. An additional enhancement might be the specification of an upper limit for the amount of memory to use.

3.16.6 Memory Handling

In addition to resource pools (which are only completely freed when a task is done), it might be useful to have memory (de)allocation routines that work in an area that is passed as parameter. This way we can restrict memory usage for a given functionality.

A debugging version of malloc/free can be taken from libsm. It might be useful to provide a version that can do some statistics, see `malloc(9)` on OpenBSD.

In case of memory allocation problems, i.e., if the system runs out of memory, the usage has to be reduced by slowing down the system or even aborting some actions (connections). The system should pre-allocate some amount of memory at startup which it can use in case of such an emergency to allow basic operation. The pre-allocated memory can either be explicitly used by specifying it as parameter for memory handling routines, or it can be simply free()d such that system operation can continue. It's not yet clear which of those both approaches is better.

3.16.7 String Abstraction

sendmail X should have a string abstraction that is better than the way C treats "strings" ('\0' terminated sequences of characters). They may be modelled after postfix's VSTRING abstraction or libsmi, the latter seems a good start. They should make use of rpools to simplify memory management. We have to analyze which operations are most needed and optimize the string abstraction for those without precluding other (efficient) usages. Some stuff:

- Append characters to the end of a string: that's ugly right now (the application itself has to keep track of the current length and the allowed length, or it has to use an inefficient `strlcat()` which looks for the string end each time).
- `snprintf()`: should be ok, nothing special here? That's dependent on whether we allow low-level operation via direct (array) access, which we probably should not do since it violates the abstraction layer. Then it comes down to the first item.
- Replace characters (or substrings) within a string: that's ugly. It should be checked whether we really need this and how it can be replaced by some library function(ality). Write at arbitrary position in string usage of string+offset as a string, e.g., s=buf+1, use s, then maybe *–s='<'. check usage: what kind of operations do we need, how often are those done, which should be efficient (and secure!)?

Note: it might be useful to have a mode for `snprintf()` that denotes appending to a string instead of writing from the beginning. Should this be a new function or an additional parameter? The former seems the best at the user level, internally it will be some flag.

Even though strings should dynamically grow, there also need to be a way to specify an upper bound. For example, we don't want to read in a line and run out of memory while doing so because some attacker feeds an "endless" line into the program.

The string structure could look like this:

```
struct sm_str_S
{
    size_t    sm_str_len;       /* current length */
    size_t    sm_str_size;      /* allocated length */
    size_t    sm_str_maxsz;     /* maximum length, 0: unrestricted? */
    uchar    *sm_str_base;      /* pointer to byte sequence */
    rpool_t   sm_str_rpool;     /* data is allocated from this */
}
```

3.16.7.1 Constant Strings

Some strings are only created once, then used (read only) and maybe copied, but never modified. Those strings can be implemented by using reference counting and a simplified version of the general string abstraction.

```
struct sm_cstr_S
{
    size_t        sm_cstr_len;       /* current length */
    unsigned int  sm_cstr_refcnt;    /* reference counter */
    uchar        *sm_cstr_base;      /* pointer to byte sequence */
}
```

Copying is then simply done by incrementing the reference counter. This is useful for identifiers that are only created once and then read and copied several times.

Available functions are:

```
/* create a cstr with a preallocated buffer */
sm_cstr_P       sm_cstr_crt(uchar *_str, size_t _len);

/* create a cstr by copying a buffer of a certain size */
sm_cstr_P       sm_cstr_scpyn(const char *_src, size_t _n);

/* duplicate a cstr */
sm_cstr_P       sm_cstr_dup(sm_cstr_P _src);

/* free a cstr */
void            sm_cstr_free(sm_cstr_P _cstr);

size_t          sm_cstr_getlen(sm_cstr_P _cstr);
bool            sm_cstr_eq(const sm_cstr_P _s1, const sm_cstr_P _s2);
```

3.16.8 Timed Events

Triggering events at certain times requires either signals or periodic checks of a list which contains the events (and their times). If we use signals, then the signal handler must be thread-safe (of course). In a multi-threaded program there is one thread which takes care of signal handling. This thread is only allowed to use a few functions (signal-safe). If the main program is event-driven, then we can use one pipe to itself and the signal handler will just write a message to it which says that a timed event occurred. In that case, the event-loop will trigger due to the I/O operation (ready for reading) and the appropriate action can be performed.

The alternative is busy-waiting (polling), i.e., use a short timeout in the main event-loop (e.g., one second) and check whenever the thread awake whether a scheduled action should be performed. This seems to be more compute-intensive than the above solution.

Another way is to set the timeout to the interval to the next scheduled event. However, this causes slight problems when new events are added in the mean time that are supposed to be performed earlier than the previous next event.

Currently the signal-handler approach seems to be the best since it provides a clean interface and it doesn't change the main event handler loop logic. Note: take a look at libisc code.

3.16.9 Shared Memory

System V shared memory can be used between unrelated processes (just share the key and give appropriate access). Other forms of shared memory (mmap()) require a common ancestor, which is available in form of the supervisor process. However, we have to be careful how such shared memory would be used (we can't create more on the fly, it's a fixed number).

3.16.10 Address Parsing

We need an RFC 2821 and an RFC 2822 parser.

RFC 2821: Routing based on address (MX) and additional rules (configuration by admin).

RFC 2822: We also need a rewrite engine to modify addresses based on rules (configuration) by the admin.

3.16.10.1 RFC 2821 parsing

The SMTP server/address resolver needs an RFC 2821 parser to analyze (check for syntactical correctness) and to decide what to do about addresses (routing to appropriate delivery agent).

Note: the syntax for addresses in the envelope (RFC 2821) and in the headers (RFC 2822) is different, the latter is almost a superset of the former. In theory we only need an RFC 2821 parser for SMTP server daemon, but some MTAs may be broken and use RFC 2822 addresses. Should we allow this? Maybe consider it as an option. This would require that we have a good library to do both where the RFC 2821 API is a subset of the RFC 2822 API.

The parser shouldn't be too complicated, the syntax is significantly simpler than RFC 2822. Quoting RFC 2821:

```
Reverse-path = Path
Forward-path = Path
Path = "<" [ A-d-l ":" ] Mailbox ">"
A-d-l = At-domain *( "," A-d-l )
      ; Note that this form, the so-called "source route",
      ; MUST BE accepted, SHOULD NOT be generated, and SHOULD be
      ; ignored.
At-domain = "@" domain
Domain = (sub-domain 1*("." sub-domain)) / address-literal
sub-domain = Let-dig [Ldh-str]
address-literal = "[" IPv4-address-literal /
                      IPv6-address-literal /
                      General-address-literal "]"
Mailbox = Local-part "@" Domain
Local-part = Dot-string / Quoted-string
      ; MAY be case-sensitive
Dot-string = Atom *("." Atom)
Atom = 1*atext
Quoted-string = DQUOTE *qcontent DQUOTE
String = Atom / Quoted-string
```

Some elements are not defined in RFC 2821, but RFC 2822; i.e., `atext`, `qcontent`.

```
IPv4-address-literal = Snum 3("." Snum)
IPv6-address-literal = "IPv6:" IPv6-addr
General-address-literal = Standardized-tag ":" 1*dcontent
Standardized-tag = Ldh-str
Snum = 1*3DIGIT  ; representing a decimal integer
      ; value in the range 0 through 255
Let-dig = ALPHA / DIGIT
Ldh-str = *( ALPHA / DIGIT / "-" ) Let-dig
IPv6-addr = IPv6-full / IPv6-comp / IPv6v4-full / IPv6v4-comp
IPv6-hex  = 1*4HEXDIG
```

```
IPv6-full = IPv6-hex 7(":" IPv6-hex)
IPv6-comp = [IPv6-hex *5(":" IPv6-hex)] "::" [IPv6-hex *5(":"
           IPv6-hex)]
      ; The "::" represents at least 2 16-bit groups of zeros
      ; No more than 6 groups in addition to the "::" may be
      ; present
IPv6v4-full = IPv6-hex 5(":" IPv6-hex) ":" IPv4-address-literal
IPv6v4-comp = [IPv6-hex *3(":" IPv6-hex)] "::"
             [IPv6-hex *3(":" IPv6-hex) ":"] IPv4-address-literal
      ; The "::" represents at least 2 16-bit groups of zeros
      ; No more than 4 groups in addition to the "::" and
      ; IPv4-address-literal may be present
```

Note about quoting: the addresses

```
<"abc"@abc.de>
<abc@abc.de>
<\a\b\c@abc.de>
```

are the same. A string is just quoted because it may contain characters that could be misinterpreted. The "value" of the string is the string without quotes. Just its representation differs. Hence the parser (scanner) must get rid of the quotes and the value must be used. The quotes are only necessary for external representation. We have to be careful when strings with "wierd" characters are used to lookup data or passed to other programs/functions that may interpret that data, e.g., a shell. The address must be properly quoted when used externally. In some cases all "dangerous" characters should be replaced for safety, i.e., a list of "safe" characters exists and each character not in that list is replaced by a safe one. Whether this replacement is reversible is open for discussion. Some MTAs just use a question mark as replacement, which of course is not reversible.

3.16.10.2 RFC 2822 parsing

The message submission program needs an RFC 2822 parser to extract addresses from headers as well from the command line. Moreover, addresses must be brought into a form that is acceptable by RFC 2822 (in headers) and RFC 2821 (for SMTP delivery).

```
atext           =       ALPHA / DIGIT / ; Any character except controls,
                        "!" / "#" /     ;  SP, and specials.
                        "$" / "%" /     ;  Used for atoms
                        "&" / "'" /
                        "*" / "+" /
                        "-" / "/" /
                        "=" / "?" /
                        "^" / "_" /
                        "`" / "{" /
                        "|" / "}" /
                        "~"
atom            =       [CFWS] 1*atext [CFWS]
dot-atom        =       [CFWS] dot-atom-text [CFWS]
dot-atom-text   =       1*atext *("." 1*atext)
```

```
qtext          =      NO-WS-CTL /      ; Non white space controls
                      %d33 /           ; The rest of the US-ASCII
                      %d35-91 /        ;  characters not including "\"
                      %d93-126         ;  or the quote character
qcontent       =      qtext / quoted-pair
quoted-string  =      [CFWS]
                      DQUOTE *([FWS] qcontent) [FWS] DQUOTE
                      [CFWS]
```

3.16.10.3 Token handling

Address parsing (RFC 2822/2821) should be based on a tokenized version of the string representation of an address. Therefore we need a token handling library (compare postfix).

3.16.10.4 Address rewrite engine

Question: can we create only one address rewriting engine or do we need different ones for RFC 2821 and 2822? Question: is it sufficient for the first version to just have a table-driven rewrite engine (only mapping like postfix, not the full rewrite engine of sendmail 8)?

3.16.11 Internal Communication

The communication between program modules in sendmail X is of course hidden inside a library. This library presents an API to the modules. In its first implementation, the library will probably use sockets for the communication.

Note: we may use "sliding windows" just as TCP/IP does to specify the amount of data the other side can receive. This allows us to send data without overflowing the recipient. Question: can we figure this out at the lower level?

3.16.11.1 Data Structures for Internal Communication

The data structures for internal communication consist of tagged fields, see also Section 3.13.12. Tagged fields are used to allow for:

- safety: the type can be checked (against the expected data),
- omission of (sub)fields: data that is not transferred must have a default value,
- compatibility: it might be possible to use modules from different versions and still let them talk to each other since the fields have unique names.

Each field has a type (probably a unique id), and a length (to avoid parsing). Each field is terminated by a 0 byte which however only exists for paranoia's sake. The length should be sufficient, field[length] must be the 0 byte. Question: do we really want to do this? We certainly don't want this for scalars (e.g., integer values). It is useful for character strings, unless we use the string abstractions (see Section 3.16.7). The entire packet has a header that contains the total length, the originator and recipient (each module in sendmail X has a unique id, which may consist of a module type identifier and a module instance identifier since there may be several instances of a module), and a protocol version.

Question: do we need a package header to identify packages? This may be required if "garbled" packages can be send over one connection. To find the begin of the next package, a linear scan for a package header would be necessary. TCP guarantees reliable connections (Unix sockets too?), so this may not be necessary. If a "garbled" package is received, the connection is closed such that the sender has to deal with the problem.

3.16.11.1.1 Buffer for Internal Communication A record communication buffer (rcb) can be realized as an extension of the string abstraction (see Section 3.16.7) by using one additional counter (rw) (or pointer) to keep track of sequential reading out of it.

```
struct sm_rcb_S
{
    size_t    sm_rcb_len;       /* current length */
    size_t    sm_rcb_size;      /* allocated length */
    size_t    sm_rcb_maxsz;     /* maximum length */
    uchar    *sm_rcb_base;      /* pointer to byte sequence */
    rpool_t   sm_rcb_rpool;     /* data is allocated from this */
    int       sm_rcb_rw;        /* read index/left to write */
}
```

Invariants:

$len \leq size$, $size \leq maxsz$

If reading: $rw < len$. If writing: $rw < 0$ (no length specified) or $rw >$ size of the data to put into the RCB.

Basic operations to create and delete RCBs:

new()	create a new rcb
free()	free an rcb

A RCB can be operated in two different modes, each of which consists of two submodes:

1. decode:
   (a) read entire record from file into rcb
   (b) decode from rcb
2. encode:
   (a) encode data into rcb
   (b) write entire rcb to file

For 1a: read record from file: if we do only record oriented operations, then we can use len to write data into the buffer and rw to keep track of how much data is left to read.

- open for receiving: rcb_open_rcv(INOUT rcb_P rcb): reset (empty) rcb for reading data into it. Set $len = 0$ and $rw = -1$.

- rcb_rcv(INOUT rcb_P rcb, IN fd_T fd, IN int rs): read data from fd to fill rcb, rs is the initial size to read.

  The first word from the file (which is the record size, see Section 3.13.12) is written into rw (only if $rw = -1$). Make sure there is enough space in the buffer (right after the size of the record is read) without exceeding the maximum size (this may require reallocation); read data from a file and put it into the buffer (asserting the size will not be exceeded), this may happen in pieces, so len and rw are updated accordingly. The buffer will be filled until $rw = 0$. Notice: the initial read may cause a problem if the number of elements to read is specified too large: it may cause more than one record to end up in the buffer.

- close after receiving: rcb_close_rcv(INOUT rcb_P rcb): close rcb after reading data into it.

For 1b: After the entire record has been copied into the buffer we can read out of it (for decoding). Decode from buffer requires the following functionality:

- open for decoding: rcb_open_dec(INOUT rcb_P rcb): reset rcb for decoding data from it; reset rw, but not len.

- read a string or an integer out of the rcb:

  rcb_getn(IN rcb_P rcb, OUT uchar *buf, IN size_t n): read n characters

  rcb_getint(IN rcb_P rcb, OUT int *val): read an integer value

  Even if we have a common encoding (e.g., lowest bit) to distingush between integers and strings we still can't use one decoding function because we need to pass a buffer and its size for strings to the function.

- close after for decoding: rcb_close_dec(INOUT rcb_P rcb).

For 2a: encoding into rcb:

- open for encoding; rcb_open_enc(INOUT rcb_P rcb, IN size_t rs): prepare rcb for encoding data into it, rs is the expected record size (0 if not known); reset len, set rw to rs.

- put an integer or a string into the buffer.

  rcb_putn(IN rcb_P rcb, IN uchar *buf, IN size_t n): put n characters into rcb at current len

  rcb_putint(IN rcb_P rcb, IN int val): put integer value into rcb at current len

  These functions check whether the expected record size is reached unless it is -1.

  Question: should we have an additional parameter: record type? Then we would just need one call (instead of three) to write (encode) one subrecord.

- close after encoding; rcb_close_enc(INOUT rcb_P rcb): write the record size into the first word (unless done at open()), reset rw.

For 2b: writing to file:

- rcb_open_snd(INOUT rcb_P rcb): open rcb for reading data from it; reset rw, but not len.

- try to write buffer to file, if it can't write the entire buffer at once: keep track of the location, write rest of buffer starting from that location on the next try.

  rcb_snd(INOUT rcb_P rcb, IN fd_T fd): read data from *rcb* and write to fd.

  Returns number of bytes left to write, i.e., > 0 if it has to be called again, $= 0$ if it is done, < 0 on error.

- rcb_close_snd(INOUT rcb_P rcb): close rcb after reading data from it.

Notice:

- The file oriented operation need to be able to operate on different file types. One of them are file descriptors, another are state threads files. Yet another one might be sm I/O files (it's not yet clear whether we can "hide" the former behind the latter).
- some of the open()/close() functions are identical in functionality. However, if we want to be paranoid, we can add a status word that tells us in which mode the rcb currently is:
  1. NONE: unused (closed or just created)
  2. RCV: receiving data from file
  3. DEC: decoding data from rcb
  4. ENC: encoding data into rcb
  5. SND: sending data to file

  Then we can check the status in the various operations and return an error if an rcb is in a wrong state. In this case, the open functions are slightly different; however, this can of course be easily controlled via a parameter instead of different functions.

The only (important) difference to the string abstraction explained in Section 3.16.7 is to keep track of sequential reading out of/writing into it. This is similar to I/O buffers (see Section 3.16.3.2).

3.16.11.2 Marking End of Record

As explained in Section 3.13.12 the end of a record should be marked. This is not only useful for data written to disk but also for data transferred over the network. Even though TCP offers a reliable connection, there is a change for receiving garbled RCBs, e.g., consider the following situation: a multi-threaded client sends RCBs to a client, one of the write attempts fails such that a partial RCB is transmitted, another thread sends an RCB whose begin will then mistaken as the end of the previous (partial) RCB. Even though the client should stop communicating with the server as soon as the timeout (write problem) occurs, an end-of-record marker will help to recognize the problem. However, the question is how to properly implement this. The best approach is to do this completely transparent to the application. *Question*: how?

3.16.11.3 A Note about Record Types

As explained in Section 3.13.12 each record entry has a type field that describes the content. There is a small problem in selecting these type because they can encode information which may as well transported separately. For example, there might be errors for A, B, C, ..., then we can have

1. multiple record types ERR-A, ERR-B, ERR-C, ..., and the error value; or

2. one record type ERR and A, B, C, ..., and the error value.

Option 1 is not easily extendible: we need to add record types for each new error and we need to recognize them on the decoding site (and obviously they have to be generated by the sender). Option 2 would cause one record type to describe a "compound" field. As long as the type of the subentries is clear this should not be a problem.

3.16.12 Common API for Storage Libraries (Maps)

There are several libraries that need to be written which provide some kind of storage functionality, e.g., hash tables, (balanced) trees, RSC (4.3.5). If we can use (almost) the same API for them, then it can be fairly easy to replace an underlying implementation to trade one storage/access method against another. This can be useful if different access methods are required. For example, while all of these provide a simple lookup function (map key to value, i.e., the typical DB functionality), some provide also additional functionality, or at least a more efficient implementation of certain access methods than others. This is exemplified by the different Berkeley DB access methods (see [OBS99] and [Sleb]), e.g., hash, btree, queue. That API might serve as an example for others that need to be implemented. Starting with version 4.2[24] Berkeley DB does support an in-memory database without a backup on disk. If this option would have been available earlier, it wouldn't have been necessary to (re-)implement some of the functionality.

Maps can be considered as a specialized version of (file) I/O: while (file) I/O is in general sequential (even though positioning can be used and some OS provide record-based I/O), map I/O is based on an access key (however, sequential access is usually provided too). It might be useful to treat maps as a subset of the generic (file) I/O layer, i.e., use its `open()`, `close()`, `read()`, `write()`, etc functions. *Todo*: investigate this further as time permits.

Note: there are basically two different kinds of maps that are used:

1. maps to store and retrieve data.

2. maps to (mostly/only) retrieve data.

The first kind is used internally by sendmail X for various purpose to store and access data. The second kind is used to control the behavior of sendmail X, e.g., to map addresses from one form into another, to change options based on various keys like the name or IP address of the site to which an connection is open, etc. Even though it might be useful to distinguish between both, it seems more generic to use one common API. Maps that don't provide write access, e.g., a map of valid users usually is not written by an MTA, simply don't have the corresponding functions calls and trying to invoke those will trigger an appropriate error.

The basic API for BDB looks like this:

[24] `DB_MPOOLFILE->set_flags()`: if `DB_MPOOL_NOFILE` is set, no backing temporary file will be opened for in-memory databases, even if they expand to fill the entire cache.

db_create	create	Create a database handle
DB->open	open	Open a database
DB->close	close	Close a database
DB->get	lookup	Get items from a database
DB->put	add	Store items into a database
DB->del	rm	Delete items from a database
DB->remove	destroy	Remove a database

This doesn't fully match the naming conventions for sendmail X explained in Section 3.16.1.2; the names for the latter are listed in the second column.

Additional functions should include `walk`: walk through all items and apply a function to them, and for some access methods a way to get elements in a sorted order out of the storage. In Berkeley DB, the former can be implemented using the cursor functions, the latter works at least if btree is used as access method.

Generic API proposal:

`_create()`	create a new object
`_destroy()`	destroy object (other name?)
`_open()`	open an object for use, this may perform an implicit `create()`
`_close()`	close an object, can't be used unless reopened, this does not destroy the object
`_reopen()`	close and open an object; this is done only if necessary, e.g., for a DB if the file on disk has changed
`_add()`	add an entry to an object
`_rm()`	remove an entry from an object
`_alloc()`	allocate an entry and add it to an object
`_free()`	free an entry and remove it from an object
`_lookup()`	lookup an entry, return the data value
`_locate()`	lookup an entry, return the entry
`_first()`	return first entry
`_next()`	return next entry

Instead of creating a new function name for each map type, the map (context) itself is passed to a generic function `map_f()`, alternatively the context can provide function pointers such that the functions are invoked as `map_ctx->f()` (as it is done by BDB, see above).

An application uses the functions as follows:

1. Initialize the map system.
2. Register map implementations.
3. Create a map instance.
4. Optionally set some options, e.g., cache size or size limits.
5. Open the map instance for use.
6. Perform map operations, e.g., lookups, additions, and removals.
7. While performing those operations maybe reopen the map based on a signal or some other event, e.g., timeout.
8. Close the map instance.
9. Optionally destroy the map, e.g., this could free all the data in the map or unlink the map file.

10. Terminate the map system.

To simplify the code the following shortcuts might be implemented:

- If _open() is passed a NULL map instance, it calls _create() internally.
- A parameter for _close() determines where it should call _destroy() the map instance too. Alternatively _destroy() could call _close() if that has not been done by the application.

3.16.12.0.1 Reloading a Map *Question*: which level should handle the _reopen() function? It could be done in the abstraction layer but then it needs to know when to reopen a map. This could be done by checking where the underlying file has changed, but this is specific to some map implementation, e.g., Berkeley DB, while a network map does not provide such a way to check. This can be solved as follows: add a flag that indicates whether the simple check (file changed) should be performed in the abstraction layer, and add a function pointer to a _reopen() function in the map implementation which will be used if the flag is not set and the function pointer is not NULL. An additional problem is that if the _open() function takes a variable number of arguments, then those need either be resupplied when _reopen() is called, or they need to be stored in the map instance[25] (which would make it necessary to preserve that data across the sequence _close(), _open() somehow), or they need to be supplied in some other way. Such "other way" could be separate initialization and option settings functions, e.g., first _create() returns a map context, then various options are set via _setoption() which is a polymorphic function (compare setsockopt(2)) (and has a corresponding map_getoption() function), and finally _open() is called with the map context that has all options set in the proper way. Closing a map then requires either two steps or at least a parameter which indicates whether to discard (destroy) the map context too. However, this requires that the underlying map implementation actually supports this; Berkeley DB does not preserve the handle across the close[26] function. One way around this is to store the options in an additional abstraction layer between the map abstraction and the map implementation, or have a variable sized array of option types and option values in the map abstraction context in addition to the pointer to the map implementation context. Then the _setoption() function could store the options in that array and _reopen() could "replay" the calls to _setoption() by going through that array. Nevertheless, the same argument can be made here: it requires that the underlying map implementation actually supports a _create() and an _open() function; if there is only one _open() function that allocates the map context and its internal data while maybe taking several arguments, then this split can not be done.

Note: it might be useful to have a _load() callback for the _open() function to initialize data in the map, e.g., to read key and data pairs from a file. This is especially useful in conjunction with _reopen() to reload a file that has changed.

3.16.12.1 Data Structure for Storage Libraries

The following data seems to be sufficient to describe elements in a storage system (e.g., database):

```
uchar     *STH_key;        /* key */
uint32_t   STH_key_len;    /* length of key */
void      *STH_data;       /* pointer to data (byte string) */
uint32_t   STH_data_len;   /* length of data */
```

[25] It is impossible in C to store a va_list.

[26] db->close() "frees any allocated resources"

An alternative is to use the string type described in Section 3.16.7. The advantage of doing so is that the functions for it can be reused.

Berkeley DB uses the following structure (called DBT) to describe elements in the database as well as keys:

```
typedef struct {
        void      *data;        /* pointer to byte string */
        u_int32_t  size;        /* length of data */
        u_int32_t  ulen;        /* length of user buffer */
        u_int32_t  dlen;        /* length of partial record */
        u_int32_t  doff;        /* offset of partial record */
        u_int32_t  flags;       /* various flags */
} DBT;
```

3.16.12.1.1 Memory Management in Storage Libraries For an internal read/write map the storage for the key itself is managed by the library or by the application, the data storage is by default managed by the application, i.e., the data must be persistent as long as the map is used. This is in contrast to BDB which manages the data itself since it has to store it on disk. The default can be changed by setting flags in the open() call to indicate whether the application supplies memory for storing the data in which case another field specifies the length of that memory section. This is done by the DEFEDB library. It is useful to provide callbacks for data allocation/deallocation[27] such that the library can (indirectly) manage the data storage if necessary, e.g., if it is supposed to automagically remove outdated entries, or if it can update an existing entry or create new entries based on whether the `add` function should allow for this, and for the `destroy` function to remove all allocated data (memory).

A map context should contain flags that describe how keys and data is handled. First a map implementation needs to specify whether the implementation can actually perform memory management functions (it may not have the required code). If it does not provide memory management function, then the application must take care of it. When a map is opened the application specifies whether keys and data are (de)allocated by the library (if available). The flags are per map instance, not per key/data pair; this would require to store the information in the key or data itself which would complicate the storage interface, e.g., it would have to encode the flags somehow and make sure that the application will only get the relevant part, not the additional control information (compare allocation contexts for `malloc(3)`).

Note: if the map abstraction uses the string type described in Section 3.16.7 then the memory management functions are those provided by the string implementation, there is no need to have additional functions and hence no function pointers are needed for this.

3.16.12.2 Abstraction Layer for Storage Libraries

An abstraction layer should be placed on top of the various storages libraries to access the latter through a generic interface. This is similar to the Berkeley DB layer which specifies which type of database to use (open). To access the underlying implementation two methods are common:

1. function pointers (e.g., Berkeley DB). In this case the storage handle contains function pointers that can be called from the application, e.g., `STH->sth_lookup()`.

[27] BDB provides a function to specify malloc(2), realloc(2), and free(2) methods.

2. dispatch functions (e.g., sendmail 8 map functions). The application calls a generic function, e.g., `map_lookup()` which receives the storage handle as one parameter. That handle in turn is used internally to invoke the correct function.

The first method avoids an additional indirection, but the second can perform generic operations in one place before/after calling the specific storage library function. This is helpful for something like an expand function (`sm_map_rewrite()`) that replaces certain parts of the result by (parts of) the key, e.g., positional parameters ("%n" in sendmail 8). Moreover, it makes it easier to deal with functions that aren't available because the wrapper can check for that (otherwise the function pointer must point to a dummy function).

The abstraction layer needs to provide functions to

- initialize and terminate itself; `sm_maps_init(sm_maps_ctx *)`, `sm_maps_terminate(sm_maps_ctx)`.
- (un)register (add/remove) a map class by name, e.g., "hash", "tree", or "ldap"; `sm_maps_add(sm_maps_ctx, sm_mapc_ctx)`, `sm_maps_rm(sm_maps_ctx, sm_mapc_ctx)`.
- lookup a map by name and return the storage handle; `sm_maps_lookup(sm_maps_ctx, str, sm_mapc_ctx *)`,

For case 2 the abstraction layer also needs to provide the generic map functions (see Section 3.16.12).

Convention: functions that operate on

- the context for all maps use `sm_maps_*(sm_maps_ctx, ...)`,
- a map class use `sm_mapc_*(sm_mapc_ctx, ...)`,
- a single map (an instance of a map class) use `sm_map_*(sm_map_ctx, ...)`.

Question: should the distinction be made clearer than just a single character?

In its simplest form the maps context contains a list or a hash table (using the type as key) of registered map classes:

```
struct sm_maps_S
{
        List/Hash sm_mapc_P;
}
```

A map class should look like this:

```
struct sm_mapc_S
{
        sm_cstr_P        sm_mapc_type;
        uint32_t         sm_mapc_flags;

        sm_map_create_F  sm_mapc_createf();
        sm_map_destroy_F sm_mapc_destroyf();
        sm_map_open_F    sm_mapc_openf();
        sm_map_load_F    sm_mapc_loadf();
```

```
        sm_map_close_F   sm_mapc_closef();
        sm_map_reopen_F  sm_mapc_reopenf();
        sm_map_add_F     sm_mapc_addf();
        sm_map_rm_F      sm_mapc_rmf();
        sm_map_alloc_F   sm_mapc_allocf();
        sm_map_free_F    sm_mapc_freef();
        sm_map_lookup_F  sm_mapc_lookupf();
        sm_map_locate_F  sm_mapc_locatef();
        sm_map_first_F   sm_mapc_firstf();
        sm_map_next_F    sm_mapc_nextf();
}
```

Flags for map classes describe the capabilities of a map class, e.g., some of these are:

MAPC-ALLOCKEY	Map can allocate storage for key
MAPC-ALLOCDATA	Map can allocate storage for data
MAPC-FREEKEY	Map can free storage for key
MAPC-FREEDATA	Map can free storage for data
MAPC-CLOSEFREEKEY	Map must free storage for key on close (destroy?)
MAPC-CLOSEFREEDATA	Map must free storage for data on close (destroy?)
MAPC-NORMAL-REOPEN	Map can be reopened using close/open

These flags indicate whether the library can perform memory management functions. If it does not set those flags, then the application must take care of it.

A map is an instance of a map class:

```
struct sm_map_S
{
        sm_cstr_P       sm_map_name;
        sm_cstr_P       sm_map_type;
        sm_mapc_P       sm_map_class;
        char           *sm_map_path;
        uint32_t        sm_map_flags;
        uint32_t        sm_map_openflags; /* flags when open() was called */
        uint32_t        sm_map_caps;   /* capabilities */
        int             sm_map_mode;
        time_T          sm_map_mtime;  /* mtime when opened */
        ino_t           sm_map_ino;    /* inode of file */
        void           *sm_map_db;     /* for use by map implementation */

        void               *sm_map_app_ctx;

        /* array of option types and option values */
        sm_map_opt_T     sm_map_opts[SM_MAP_MAX_OPT];
}
```

There are different type of flags for maps: some describe the state of the map, some describe the functionalities (capabilities) that are offered. Capabilities are also something that can be requested when a map is opened, if a map does not offer the requested functionality, the open fails. Note: it might be useful to have a query function that returns the capabilities of a map class.

Flags that describe the state of a map are: (some of these can probably be collapsed if locking is used)

CREATED	Map has been created
INITIALIZED	Map has been initialized
OPEN	Map is open
OPENBOGUS	open failed, do not call close
CLOSING	map is being closed
CLOSED	map is closed
VALID	this entry is valid
WRITABLE	open for writing
ALIAS	this is an alias file

Generic flags (options) that describe how the map abstraction layer should handle various operations:

INCLNULL	include nul byte in key
OPTIONAL	don't complain if map can't be opened
NOFOLDCASE	don't fold case in keys
MATCHONLY	don't use the map value
ALIAS	this is an alias file
TRY0NUL	try without nul byte
TRY1NUL	try with nul byte
LOCKED	this map is currently locked
KEEPQUOTES	don't dequote key before lookup
NODEFER	don't defer if map lookup fails
SINGLEMATCH	successful only if match returns exactly one key

The `alloc()` and `free()` functions receive the application context as additional parameter:

```
typedef void *(*sm_map_*_alloc_F)(size_t size, void *app_ctx);
typedef void (*sm_map_*_free_F)(void *ptr, void *app_ctx);
```

The `create()` function creates a map instance:

```
typedef sm_ret_T (*sm_map_create_F)(sm_mapc_P mapc, sm_cstr_P name, sm_cstr_P type, uint32_t
flags, sm_map_P *pmap, ...);
```

The `open()` function opens a map for usage:

```
typedef sm_ret_T (*sm_map_open_F)(sm_mapc_P mapc, sm_cstr_P name, sm_cstr_P type, uint32_t flags,
char *path, int mode, sm_map_P *pmap);
```

Question: what is the exact meaning of the parameters, especially `name`, `flags`, `path`, and `mode`? `name`: name of the map, can be used for socket map? This is mostly a descriptive parameter. `path`: if there is a file on disk for the map this is the name of that file. So what are `flags` and `mode`? `mode` could have the same meaning as the mode parameter of the Unix system call `chmod(2)`. `flags` could have the same meaning as the flags parameter of the Unix system call `open(2)`, e.g.,

O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_NONBLOCK	do not block on open
O_APPEND	append on each write
O_CREAT	create file if it does not exist
O_TRUNC	truncate size to 0
O_EXCL	error if create and file exists
O_SHLOCK	atomically obtain a shared lock
O_EXLOCK	atomically obtain an exclusive lock
O_DIRECT	eliminate or reduce cache effects
O_FSYNC	synchronous writes
O_NOFOLLOW	do not follow symlinks

Whether these flags actually make sense depends on the underlying map implementation. *Questions*:

- Can we specify a superset of the flags, i.e., the union of all flags for all map implementation, or should those flags be map specific?
- Maybe there should be a common set that makes sense for most map types (e.g., RDONLY, NONBLOCK, CREAT) and then map specific flags that could even overlap?
- What should happen if a flag cannot be honored by a map implementation? For some it makes sense to return an error (e.g., a read-only map cannot be open in write mode), for others it might be ok to simply ignore the flags.

Here is an example of the valid flags for a map implementation (Berkeley DB), some of which have equivalents from `open(2)`:

DB_AUTO_COMMIT	
DB_CREATE	O_CREAT
DB_EXCL	O_EXCL
DB_DIRTY_READ	
DB_NOMMAP	
DB_RDONLY	O_RDONLY
DB_THREAD	
DB_TRUNCATE	O_TRUNCATE

The `load()` function needs the map context and the name of the file from which to read the data:

```
typedef sm_ret_T (*sm_map_load_F)(sm_map_P map, char *path);
```

The `close()` function requires only the map context:

```
typedef sm_ret_T (*sm_map_close_F)(sm_map_P map);
```

Adding an item requires the map context, the key, and the data (note: see elsewhere about whether the map needs to copy key or data).

```
typedef sm_ret_T (*sm_map_add_F)(sm_map_P map, sm_map_key key, sm_map_data data, unsigned int
flags);
```

Removing an item requires the map context and the key:

```
typedef sm_ret_T (*sm_map_rm_F)(sm_map_P map, sm_map_key key);
```

```
typedef sm_ret_T (*sm_map_alloc_F)(sm_map_P map, sm_map_entry *pentry);
```

`typedef sm_ret_T (*sm_map_free_F)(sm_map_P map, sm_map_entry entry);`

`typedef sm_ret_T (*sm_map_lookup_F)(sm_map_P map, sm_map_key key, sm_map_data *pdata);`

`typedef sm_ret_T (*sm_map_locate_F)(sm_map_P map, sm_map_key key, sm_map_entry *pentry);`

`typedef sm_ret_T (*sm_map_first_F)(sm_map_P map, sm_map_cursor *pmap_cursor, sm_map_entry *pentry);`

`typedef sm_ret_T (*sm_map_next_F)(sm_map_P map, sm_map_cursor *pmap_cursor, sm_map_entry *pentry);`

Notes:

- if some map types have wildly varying options or arguments then it might be necessary to provide individual `sm_map_parse_args)()` functions to deal with those differences.
- some functions need flags, e.g., `sm_map_add()`, which require that the abstraction layer defines a superset of all flags and the individual implementations translate the common flags into their individual representations. In some cases the flags are a combination of values and bitfields; this is at least the case for Berkeley DB.
- it might be necessary to have an "intermediate" structure that contains only a pointer to `sm_map_T` and a reference count if a map can be registered under several names.

```
struct sm_mapn_S
{
        sm_map_P        sm_mapn_map;
        uint32_T        sm_mapn_refcnt;
}
```

3.16.12.3 Dynamic (Re)Open of Maps

sendmail 8 opens map dynamically on demand. This can be useful if some maps are not needed in some processes. Moreover, some map types do not work very well when shared between processes. For sendmail X it can be useful to close an reopen maps in case of errors, e.g., network communication problems. *Question*: which level should handle this: the abstraction layer or the implementation? If it's done in the abstraction layer then code duplication can be avoided, however, it's not yet clear what the proper API for this functionality is. It seems that using `sm_map_close()` and `sm_map_open)()` is not appropriate because we already have the map itself. Would `sm_map_reopen()` (see Section 3.16.12.0.1) be the appropriate interface?

3.16.12.4 Asynchronous Operation of Maps

The previous sections describe a synchronous map interface. Section 3.1.1 contains a generic description of asynchronous functions. The map abstraction layer can either provide just a synchronous map API, or it can provide the more generic approach described in Section 3.1.1, however, it should let the upper layer "see" whether the lower level implements synchronous or asynchronous operations.

3.16.12.5 Multi-Threading

In most cases access to maps must be protected by mutexes. This can be either provided by the abstraction layer or by the low-level implementation. Doing it in the former minimizes the amount of code duplication.

3.16.12.6 Multiple Map Instances

Some map implementations may be slow (e.g., network based lookups) and hence some kind of enhancement is required. One way to do this is to issue multiple requests concurrently. Besides asynchronous lookups (which are currently not supported, see Section 3.16.12.4) this can be achieved by having multiple map instances of the same map type provided the underlying implementation allows for that.

It would be very useful if this can be provided by the map abstraction layer to avoid having this implemented several times in the low level map types.

Question: how to do this properly? The map type needs to specify that it offers this functionality and some *useful* upper limit for the number of concurrent maps[28].

3.16.13 DNS

Unfortunately older DNS resolvers (BIND 4.x) are not thread safe. The resolver that comes with BIND 8 (and 9) is supposed to be thread safe, but even FreeBSD 4.7 does not seem to support it (`res_nsearch(3)` is not in any library). In general there are two options to use DNS:

1. use a thread safe resolver, e.g., by including the necessary parts of BIND 8, and create several threads as DNS resolvers which may block.

2. do it yourself: write an asynchronous DNS resolver. Use `res_mkquery()` to construct a DNS query and send it to a DNS server. If the event thread library (see Section 3.20.5) is used then a single task can be used for the communication with the DNS server. Several tasks may be started to communicate with several DNS servers. Moreover, there is the slight complication of using UDP or TCP, which may require yet another (set of) task(s).

Option 1 does not directly work for applications that use state threads (see Section 3.20.3.1), the communication must at least be modified to use state threads I/O to avoid blocking of the entire application. Therefore it is probably not much different to chose option 2, which would be a customized (and hopefully simpler) version of the resolver library. This option however causes a small problem: how should the caller, i.e., the task that requested a DNS lookup, be informed about the result? See Section 3.1.1.1 for a general description of this problem. We could integrate the functionality directly into the application; e.g., for an application which uses event threads this is exactly the activation sequence that is used: a request is send, the task waits for an I/O event and then is activated to perform the necessary operations. However, this tightly integrates the DNS resolver functionality into the application which may not be the best approach. Alternatively the DNS request can include a context and a callback which is invoked (with the result and the context) by the DNS task. The latter is useful for an application using the event thread library. For a state threads application a different notification mechanism is used, i.e., a condition variable. See Section 4.1.2 for a description of the possible implementations.

A DNS request contains the following elements:

[28] Specifying one as limit means that this is a single instance map only, hence no additional flag is necessary.

name	hostname/domain to resolve
type	request type (MX, A, AAAA, ...)
flags	flags (perform recursion, CNAME, ...)
dns-ctx	DNS resolver context
event thread:	
app-ctx	application context
app-callback	application callback
state threads:	
app-cond	condition variable to trigger

The DNS resolver context is created by a `dns_ctx_new()` function and contains at least these elements:

DNS servers	list of DNS servers to contact
flags	flags (use TCP, ...)

Available functions include:

dns_mgr_ctx_new()	create a new DNS manager context
dns_mgr_ctx_free()	free (delete) DNS manager context
dns_rslv_new()	create a new DNS resolver context
dns_rslv_free()	free (delete) DNS resolver context
dns_tsk_new()	create a new DNS task
dns_tsk_free()	free (delete) DNS task
dns_req_add()	send DNS request; this may either receive a DNS request structure as parameter or the individual elements

There should be only one DNS task since it will manage all DNS requests. However, this causes problems due to:

- Multiple DNS servers: these need different tasks, at least in the event thread model due to the I/O activation.
- UDP vs TCP communication: different file descriptors need different tasks (in the event thread model, see above).

If multiple DNS resolver tasks are used then there is the problem of distributing the requests between them and of a fallback in case of problems, e.g., timeout or truncations (UDP).

A DNS resolver task should have the following context:

DNS server	which server? (fd is stored in task context)
flags	flags (UDP, TCP, ...)
dns-ctx	DNS resolver context
error counters	number of timeouts etc

If too many errors occurred the task may terminate itself after it created a new task using a different DNS server.

3.16.14 Timeouts

Many functions can cause temporary errors of which timeouts, especially from asynchronous functions, are hard to handle. Question: should the caller implement a timeout or the callee? That is, should the caller set some timestamp on its request and the data where the result should be stored and check this periodically or should the callee implement the timeout and guarantee returning a result (even if it is just an error code) to the caller? The latter looks like the better solution because it avoids periodic scanning in the application (caller), however, then it needs to be in the callee, which at least has some centralized list of outstanding requests. The caller has to make sure that its open requests are removed after some time, e.g., some sort of garbage collection is required. If the caller removes a request but the callee returns a result later on, then the caller must handle this properly; in the simplest case it can just ignore the result. A more sophisticated approach would use a `cancel_request()` function such that the request is also discarded by the callee.

In some cases the caller can easily implement a timeout, e.g., SMTPS does this when it waits for a reply from QMGR. If the reply does not arrive within a time limit, then the SMTP server returns a temporary error code to the client.

3.16.15 Misc

Todo: structure this.

- Sockets (fd) can be passed to other (unrelated) processes using `sendmsg()` etc., see [Ste92]. One listener process – another smtp server? How to do that on NT? According to someone who knows NT it can be done.

- event library: should be a fairly abstract model so we can use select(), poll(), kqueue(), I/O completion? Note: take also a look at libisc code and the MMA.

- thread library (NT/Unix),

- I/O library, see libsm/. Must made NT compatible, no extra NT layer underneath. More functionality: no read/write in sendmail, sm_* instead, etc. write/read/fsync/etc with "please do it till", so the library can order and group these functions

  For example: after final dot a message must (probably) be fsync()ed, but we don't want to do this for each individually, but maybe a group. Then we use fsync(within-10-seconds) and the library can group several several requests together (compare softupdates).

- I/O buffering: how about a ring buffer? Make it twice as big as the maximum size, so it can be accessed as an array. If we write beyond the middle (the "official" end), we must write the data twice: middle + x and start + x. This way we can access the buffer without worrying about wrap around. Advantage: fast read access. Disadvantage: ugly for writing, twice as much space.

- milter_read/write: almost the same in mta and milter? functions to open sockets based on textual specifications: similar/same in many programs → library generic listener? (compare mapd, milter, libsmi)

- Generic interface via sockets: define a generic protocol similar to milter. make sure it's extensible (version, offered options, required options, i.e., option negotation)

- Configuration parser: the parser of the mta is ugly...

- Asynchronous DNS library Note: take a look at lwres in BIND 9.

3.16.16 Logging

3.16.16.1 ISC Logging

ISC [ISC01] logging uses a context (type `isc_log_t`), a configuration (type `isc_logconfig_t`), categories and modules, and channels.

Channels specify where and how to log entries:

```
<channel> ::= "channel" <channel_name> "{"
     ( "file" <path_name>
         [ "versions" ( <number> | "unlimited" ) ]
         [ "size" <size_spec> ]
       | "syslog" <syslog_facility>
       | "stderr"
       | "null" );
     [ "severity" <priority>; ]
     [ "print-category" <bool>; ]
     [ "print-severity" <bool>; ]
     [ "print-time" <bool>; ]
   "}";
<priority> ::= "critical" | "error" | "warning" | "notice" |
               "info" | "debug" [ <level> ] | "dynamic"
```

For each category a logging statement specifies where to log entries for that category:

```
<logging> ::= "category" <category_name> { <channel_name_list> };
```

Categories are "global" for a software package, i.e., there is a common superset of all categories for all parts of the package. Some parts may only use a subset, but the meaning of a category must be consistent across all parts, otherwise the logging configuration will cause problems (at least inconsistencies). Modules are use by the software (libraries etc) to describe from which part of the software a logging entry has been made.

The API is as follows:

- Create and delete a logging context:

  ```
  isc_log_create(isc_mem_t *mctx, isc_log_t **lctxp, isc_logconfig_t **lcfgp);
  isc_log_destroy(isc_log_t **lctxp);
  ```

- To manipulate logging configurations these functions are available:

  ```
  isc_logconfig_create(isc_log_t *lctx, isc_logconfig_t **lcfgp);
  isc_logconfig_use(isc_log_t *lctx, isc_logconfig_t *lcfg);
  isc_logconfig_get(isc_log_t *lctx);
  isc_logconfig_destroy(isc_logconfig_t **lcfgp);
  ```

- To register categories and modules:

```
isc_log_registercategories(isc_log_t *lctx, isc_logcategory_t categories[]);
isc_log_registermodules(isc_log_t *lctx, isc_logmodule_t modules[]);
```

- To create and use channels:

```
isc_log_createchannel(isc_logconfig_t *lcfg, const char *name,
      unsigned int type, int priority,
      const isc_logdestination_t *destination,
      unsigned int flags)
isc_log_usechannel(isc_logconfig_t *lcfg, const char *name,
   const isc_logcategory_t *category,
   const isc_logmodule_t *module)
```

- To actually write a logfile entry:

```
isc_log_write(isc_log_t *lctx, isc_logcategory_t *category,
      isc_logmodule_t *module, int priority, const char *format, ...)
isc_log_vwrite(isc_log_t *lctx, isc_logcategory_t *category,
       isc_logmodule_t *module, int priority,
       const char *format, va_list args)
```

- Miscellaneous functions:

```
isc_log_setdebuglevel(isc_log_t *lctx, unsigned int level);
isc_log_getdebuglevel(isc_log_t *lctx);

isc_log_wouldlog(isc_log_t *lctx, int priority);

isc_log_setduplicateinterval(isc_logconfig_t *lcfg, unsigned int interval);
isc_log_getduplicateinterval(isc_logconfig_t *lcfg);

isc_log_settag(isc_logconfig_t *lcfg, const char *tag);
isc_log_gettag(isc_logconfig_t *lcfg);

isc_log_opensyslog(const char *tag, int options, int facility);
isc_log_closefilelogs(isc_log_t *lctx);

isc_log_categorybyname(isc_log_t *lctx, const char *name);
isc_log_modulebyname(isc_log_t *lctx, const char *name);
```

3.17 Modules

There are some technical problems that must be solved by a clean design. For example, in sendmail 8.12 the function that returns error messages for error codes includes the basic OS error codes but also extensions like LDAP if it has been compiled in. This makes it hard to reuse that library function in different programs that don't want to use LDAP because it will be linked in since it is referenced in the error message function. This must be avoided for obvious reasons.

One possible solution is to have a function list where modules register their error conversion functions. However, this requires that the error codes are disjunct. This can be achieved as explained in Section 3.15.2. So the current approach (having a "global' error conversion function) doesn't work in general. The error conversion must be done locally and the error string must be properly propagated.

3.18 Building sendmail X

Question: how do we build sendmail X on different systems? Currently we have one big file (`conf.h`) with a lot of OS-specific definitions and several OS-specific include files (`os/sm_os_`*OS*`.h`). However, this is ugly and hard to maintain. We should use something like `autoconf` to automagically generate the necessary defines on the build system. This should minimize our maintainance overhead, esp. if it also tests whether the feature actually works.

3.18.1 autotools

Since we do not have enough man power to develop yet another build system and since our current system is completely static, we will use the GNU autotools (automake, autoconf, etc) for sendmail X. We already have a (partial) patch for sendmail 8 to use autoconf etc (contributed by Mark D. Roth). We can use that as a basis for the build configuration of sendmail X.

Interestingly enough, most other open source MTAs use their own build system. However, BIND 9 and Courier MTA use also autoconf.

Note: there are some things which can't be solved by using autoconf. Those are features of an OS that cannot be (easily or at all) determined by a test program. Examples for these are:

- Stack growth direction (required by state threads).
- Inheritance of non-blocking state through accept(), i.e., if the socket on which connections are accepted is set to non-blocking, will the new socket be also non-blocking?
- Serialize accept() (see state threads).
- Anonymous memory mapping is slightly different on SVR4 and BSD4.3 derived platforms.

In those case we need to provide the data in some script (config.cache?) that can be easily used by configure. The data should probably be organized according to the output of config.guess: "CPU-VENDOR-OS", where OS can be "SYSTEM" or "KERNEL-SYSTEM".

3.19 Operating System Calls

This section contains hints about some operating system calls, i.e., how they can be used.

3.19.1 General Hints

This section should contain only standard (POSIX?) behavior, nothing specific to some operating system. See the next section for the latter.

Todo: structure this and then use it properly.

- Socket I/O: modern OS can use SO_SNDTIMEO, SO_RCVTIMEO to impose timeouts.
- accept(2):

> (OpenBSD man page, should apply to all OS) If the socket is marked non-blocking and no pending connections are present on the queue, accept() returns an error as described below.
>
> It is possible to select(2) or poll(2) a socket for the purposes of doing an accept() by selecting it for read.
>
> One can obtain user connection request data without confirming the connection by issuing a recvmsg(2) call with an msg_iovlen of 0 and a nonzero msg_controllen, or by issuing a getsockopt(2) request. Similarly, one can provide user connection rejection information by issuing a sendmsg(2) call with providing only the control information, or by calling setsockopt(2).

Question about the last sentence: how to do this and is it portable?

3.19.2 Operating System Specific Hints

This section contains hints about the behavior of some system calls on certain operating system. That is, anything that is interesting with respect to the use of system calls in sendmail X. For example: don't use call xzy in these situations on this OS.

Todo: structure this.

- select(2): OpenBSD (2.8): Internally to the kernel, select() works poorly if multiple processes wait on the same file descriptor.

- FreeBSD: send(2)/recv(2) can be used to obtain the user-id of a process that is connected to a local socket. This would be useful to allow mail submission via a local socket and obtain the user credentials. However, FreeBSD seems to be the only OS that supports this.

3.20 Worker Threads

Just some unsorted notes about worker threads.

We have a pool of worker threads. Can/should this grow/shrink dynamically? In first attempt: no, but the data structures should be flexible. We may use fixed size arrays because the number of threads will possibly neither vary much nor will it be very large. However, fixed size doesn't refer to compile time, but configuration (start) time.

We have a set of tasks. This set will vary (widely) over time. It will have a fixed upper limit (configuration time), which depends for example on the number of available resources of the machine/OS, e.g., file descriptors, memory.

We need a queue of runnable tasks (do we need/want multiple queues? Do we want/need prioritized queues?). Whenever a task becomes active (in our case usually: I/O is possible) then it will be added to the "run" queue (putwork()). A thread that has nothing to do, looks for work by calling getwork(). These two functions access the queue (which of course is protected by a condition variable).

Maybe we can create some shortcuts (e.g., if a task becomes runnable and there is an inactive worker thread, give it directly to it), but that's probably not worth the effort (at least in the first version).

3.20.1 SMTP Server and Worker Threads

A SMTP server is responsible for a SMTP session. As such it has a file descriptor (fd) that denotes the connection with the client. The server reads commands from that fd and answers accordingly. This seems to be a completely I/O driven process and hence it should be controlled by one thread that is watching I/O activities. This assumes that local operations are fast enough or that they can be controlled by the same loop.

There is one control thread that does something like this:

```
while (!stop)
{
  check for I/O activity
  - timeout? check whether a connection has really timed out
     (inactive for too long)
  - input on fd:
     lookup context that takes care of this fd
     add context to runnable queue
  - fd ready for output:
     lookup context that takes care of this fd
     add context to runnable queue
  - others?
}
```

Question: do we remove fd from the watch-set when we schedule work for it? If so: who puts it back? A worker thread when it performed the work?

The worker threads just do:

```
while (!stop)
{
  get_work(&ctx);
  ctx->perform_action(ctx);
}
```

See also Section 3.16.4 about programming problems.

3.20.2 Worker Thread Library

It would be nice to build a skeleton library that implements this model. It then can be filled in with the appropriate data structure, that contain (links to) the actual data that is needed.

It would be even better to implement the generic model (multiple, pre-forked processes with worker threads) as a skeleton. This should be written general enough such that it can be tweaked (run time/compile time) to extreme situations, i.e., one process or one thread. The latter is probably not possible with this model, since there must be at least one control thread and one worker thread. Usually there's also a third thread that deals with signals.

However, this generic model could be reused in other situations, e.g., the first candidate would be libmilter.

3.20.3 Thread Libraries

This section contains some comments about available thread libraries. We try to investigate whether those libraries are suitable for use in sendmail X, and if so, for which components.

3.20.3.1 State Threads for Internet Applications

Some comments about [SGI01]: State Threads (ST) for Internet Applications (IA).

> We assume that the performance of an IA is constrained by available CPU cycles rather than network bandwidth or disk I/O (that is, CPU is a bottleneck resource).

This isn't true for SMTP servers in general, they are disk I/O bound. Does this change the suitability of state threads for SMTPS?

> The state of each concurrent session includes its stack environment (stack pointer, program counter, CPU registers) and its stack. Conceptually, a thread context switch can be viewed as a process changing its state. There are no kernel entities involved other than processes. Unlike other general-purpose threading libraries, the State Threads library is fully deterministic. The thread context switch (process state change) can only happen in a well-known set of functions (at I/O points or at explicit synchronization points). As a result, process-specific global data does not have to be protected by mutual exclusion locks in most cases. The entire application is free to use all the static variables and non-reentrant library functions it wants, greatly simplifying programming and debugging while increasing performance.

The application program must be extremely aware of this! For example: x = GlobalVar; y = f(x); GlobalVar = y; is "dangerous" if f() has a "synchronization point".

Note: Any blocking call must be converted into an I/O event, otherwise the entire process will block, because scheduling is based on asynchronous I/O. This doesn't happen with POSIX threads. Does this make ST unusable for the SMTPS? For example, fsync() may cause a problem. Question: can we combine ST and POSIX threads? The latter would be used for blocking calls, e.g., fsync(), maybe read()/write() to disk, or compute-intensive operations, e.g., cryptographic operations during TLS handshake. Answer: no [She01a].

Note: if you link with a library that does network I/O, it must use the I/O calls of ST [She01b]:

> This is a general problem – external libraries should conform to the core server architecture. E.g., if the core server uses POSIX threads, all libraries must be thread-safe and if the core server is ST-based, all libraries must use ST socket I/O.

That might be an even bigger problem than the compute-intensive operations. However, those libraries might only be used in the address resolver.

3.20.4 Dealing with Blocking Sections for Worker Threads

Since purely event based programming is hard as explained in Section 3.16.4, other approaches have been suggested.

The problems with a programming model that uses one thread for each connection have already been mentioned in Section 2.5.2, 4a. Additionally, the scheduling of those threads is entirely up to the OS (or the thread library).

We would like to reduce the number of threads without having to resort to the complicated event based programming model. Hence we use a worker model, but we do not need to split functions when they perform a blocking call. Instead, we add some mechanism such that the worker thread library "knows" whenever a thread executes those (compute intensive or blocking) functions. This "helps" the OS/thread library to schedule threads by restricting the number of threads in those sections (regions).

3.20.4.1 Worker Threads with Semaphores

One way to deal with the problem explained above is to protect those regions by counting semaphores to limit the number of threads that can execute those (compute intensive or blocking) functions.

Example: we have one counting semaphore (iosem) for events that must be taken care of and one (compsem) for a compute intensive section. Before a worker thread takes a task out of the queue in which the event scheduler added them, it must acquire iosem. When a thread wants to enter the compute intensive section, it releases iosem (thus another worker thread can take care of an event task), and acquires compsem, which may cause it to block (thus allowing another thread to run). After the thread finished the compute intensive section, it releases compsem and acquires iosem again before continuing.

Notice: it is still possible that the OS scheduler will let a compute intensive operation continue without switching to another thread. Scheduling threads is usually cooperative, not time-sliced. Hence we may have a similar problem here as for state threads.

3.20.4.2 Worker Threads with Dynamic Limits

Another way to deal with tasks that can block is to increase the number of allowed threads dynamically whenever a task enters a code section in which it may block. When it leaves that code section, the number of allowed threads is decreased again.

Note: this looks like a very simple solution to deal with threads that can block, however, there are some details that need to be handled properly. For example, there can be many threads which wait for an event and hence each of them increases the number of allowed threads such that the system can continue to operate. Next all of the threads can be unblocked because the events for which they are waiting occurred — e.g., the DNS library answers requests for the same item all at once — hence the number of allowed threads is decreased significantly, such that there are more current threads than allowed threads. If the tasks that have been waiting for the events terminate almost immediately, e.g., just prepare the results for sending back to the clients, then the worker threads that executed those tasks become idle, but the number of current tasks exceeds the allow limit. Hence the scheduling of new tasks must not just take the number of current tasks and the allowed number of tasks into account but also the number of idle tasks that can be reused. However, instead of checking whether the number of idle tasks is greater than zero the value should be compared with the number of allowed tasks. The intention is that the number of active tasks is less than the number of allowed tasks.

3.20.4.3 Counting Active Threads

Instead of changing the limits of allowed threads, it might be simpler to count the number of active threads, especially considering the last paragraph in the previous section. Initially the number of active threads is obviously zero. Whenever a worker executes a task, the number is increased, and decreased then the (user) function returns. The number of active threads is decreased whenever a task enters a code section in which it blocks, and increased after it unblocks.

New tasks can be scheduled if the number of active tasks (instead of the number of total/current tasks) is less than the number of allowed tasks. This check is easier than the one described in the previous section.

3.20.4.4 Summary: Dealing with Blocking Sections for Worker Threads

Only the proposal explained in Section 3.20.4.1 actually restricts the number of threads that may perform a compute intensive function. None of the proposals however restricts the number of threads because there can always be threads that are blocked waiting to acquire a semaphore or waiting for a condition. To avoid this, functions must actually be written in a different style (as "continuations"). However, even that approach does not solve all the problems because in some cases it is impossible to know at the application level whether a (library) function may block.

3.20.5 Event Thread Library

An event thread library provides the basic framework for a worker based thread pool that is driven by I/O events and by wakeup times for tasks.

3.20.5.1 Overview

The library uses a general context that describes one event thread system and a per task context. It maintains two queues: a run queue of tasks that can be executed, and a wait queue of tasks that wait for some event (IS-READABLE, IS-WRITABLE, TIMEOUT, NEW-CONNECTION (listen())). Each task is at most in one of the queues at each time; if it is taken out of a queue, it is under the sole control of the function that removed it from the queue. If this model can be achieved then no per-task mutex is required, i.e., access to a task is either protected by the mutex for the queue it is in or is is "protected" by the fact that it isn't in a queue.

3.20.5.1.1 Worker Threads The system maintains some number of worker threads between the specified minimum and maximum. Idle threads vanish after some timeout until the minimum number is reached. An application first initializes the library (`evthr_init()`), then it creates at least one task (`evthr_task_new()`), and thereafter calls `evthr_loop()` which in turn monitors the desired events (I/O, timeout) and invokes the registered callback functions. Those callback functions return a result which indicates what should happen next with the task:

flag	meaning
OK	do nothing, task has been taken care of
WAITQ	put in wait queue
RUNQ	put in run queue
SLPQ	sleep for a while
DEL	delete task
TERM	terminate event thread loop

3.20.5.1.2 Changing Event Flags Additionally the task may want to change the events it is waiting for. This can be accomplished in several ways:

1. Direct modification of the event flags in the task; this is feasible if the access model given above can be achieved.

2. Return a value to indicate how to change the event flags.

3. Provide a function that changes the event flags while it has proper access to the task. Notice: this does not necessarily mean that the task must be locked via a mutex, it just means that no other thread reads or writes the event flags when the function changes it.

Solution 2 would extend the return values given before by more flags (the required values can be easily encoded as bits in an integer value). These flags could be returned as "SET" and "CLEAR".

flag	meaning
RD	IS-READABLE
WR	IS-READABLE
SL	TIMEOUT
LI	NEW-CONNECTION

3.20.5.1.3 Maximize Concurrency To maximize concurrency, the system must be able to handle multiple requests over one file descriptor concurrently. This is required for the QMGR if it serves requests from SMTPS, which is multi-threaded and will use only one connection per process, over which it will multiplex all requests to (and answers from) the QMGR. Hence there must a (library supplied) function that can be called from an application to put a task back into the wait queue after it read a request (usually in form of an RCB, see Section 3.16.11.1.1). Then the thread can continue processing the request while the task manager can wait for an event on the same file descriptor and schedule another thread to take care of it. This way multiple requests can be processed concurrently.

3.20.5.1.4 Changing Event Types: Problem This raises another problem: usually tasks in a server wait for incoming requests. Then they process them and send back responses. However, if an application returns a task as soon as possible to the wait queue, then it can't change the event types for the task (to include IS-WRITABLE so the answer is sent back to the client), because it relinquished control of it. So either the task descriptions must be protected by a mutex (which significantly complicates locking and error handling), or a different way must be used to change the event mask of a task (see 3.20.5.1.2). One such way is to provide a function that informs the task manager of the change of the event mask, i.e., send the task manager (over an internal communication mechanism, usually a pipe) the request to enable a certain event type (e.g., IS-WRITABLE). Since the task manager locks both queues as soon as an event occurred, it can easily change the task event type without any additional overhead. Moreover, this solution has the useful side effect that the wait queue is scanned again and the new event type is added to the list of events to wait for. However, the task might be active, i.e., in neither of the queues. In that case, we can either request that the user-land program does not change the status flags (which is probably a good idea anyway), or we actually have to use a mutex per task, or we need to come up with another solution.

A related problem is "waking" up another task. For example, we may have a task that periodically performs a function if data is available on which to act. If no data is available, then this task should not wake up at all, or only in long intervals. Initially the task may wake up in long intervals, look for data, and go back to sleep for the long interval if no data is available. If however data becomes available while the task is sleeping for the long interval, its timeout should be decreased to the short interval. There might be varying requirements:

1. perform the function immediately if data becomes available. In that case the task that makes the data available might perform that function itself, unless concurrency is possible.
2. perform the function only in certain (short) intervals if data is available. Group commits are a perfect example for this (which triggered this discussion here).

This can probably be handled similar to the problem mentioned before, in addition to telling the task manager that a certain event type should be enabled, it also tells it the new timeout.

3.20.5.1.5 Changing Event Types: Discussion The task context should contain the following status flags:

1. what kind of events the task is waiting for (event request flags). These flags may be modified by the application and by the library.
2. what kind of events did occur (event occurred flags). These flags are only modified by the library and read by the application.
3. internal status/control flags. These flags are only accessed by the library, never by the application.

There doesn't seem to be a way around using a mutex to protect access to the event request flags (1) and similar data, e.g., the timeout of a task. If some data can be modified in a task context while it is not "held" because it's in one of the queues, then we need to protect it.

Maybe we can split the event request flags:

1. current event request flags, which are only modified by the library.
2. new event request flags, which are modified by the application and the library.

Then we can modify the event request flags of a task when it is "under control", i.e., when it is in a queue and the main control loops examines the tasks. That is, a function adds a change request for a task to a list; the change request includes the new event request flags and the task identifier (as well as other data that might be necessary, e.g., new sleep time). When the main loop is invoked, it will check the change request list and apply all changes to tasks that are under its control, i.e., in the wait queue. This should allow us to avoid per-task mutexes.

Alternatively, we can use mutexes just to protect the "new request" flags (see list above: item 2). That way the protected data is fairly small and there should almost never be any contention. Moreover, we can avoid having a list (with all its problems: allocation, searching for data, etc).

3.20.5.1.6 Idle Task It might be useful to be able to specify a task that is invoked if the system is idle (the so-called "idle task"). For example, if the system doesn't do anything (all threads idle), invoke the idle task. That task may take some options like:

- how long must the system be idle before this is invoked?
- what's the minimum time between invocations?

Question: how to "signal" the idle task when the system is busy again? It might be useful to give this task a low priority in case other tasks are started while it is running such that the scheduler can act accordingly. Notice that some pthread implementation do not care much about priorities, some even implement cooperative mult-threading.

3.20.5.1.7 Misc Question: what about callback functions for signals?

Chapter 4

Sendmail X: Implementation

This chapter describes some parts of the current sendmail X implementation. Obviously the implementation may change at any time, this is not part of the sendmail X specification. Only the external interfaces, i.e., those described in previous chapters, specify the sendmail X behavior and configuration. The usual disclaimer "may be subject to changes" applies, i.e., nobody should rely on the descriptions given here, they are for people who actually want to understand (and modify or maintain) the source code. Some of the descriptions might not be up to date (usually the source code is implemented/changed first and updating the documentation may lag a bit).

4.1 Introduction to the Sendmail X Source Code

There are some parts of the sendmail X source code which should be explained before the individual modules and the libraries. Those are conventions that are common to at least two modules and which will be used in the following sections.

4.1.1 Identifiers

In Section 3.1.1.2 some remarks about identifiers have been made. In this section the structure of session and transaction identifiers are explained. These identifiers for the SMTP server consist (currently) of a leading 'S', a 64 bit counter, and an 8 bit process index, i.e., the format is: `"S%016qX%02X"`. Those identifiers are of course used in other modules too, especially the QMGR. For simplicity the identifiers for the delivery agents follow a similar scheme: a leading 'C', an 8 bit process index, a running counter (32 bit) and a thread index (32 bit). Notice that only the SMTPS identifiers are supposed to be unique over a very long time period (it takes a while before a 64 bit counter wraps around). The SMTPC (DA) identifiers are unique for a shorter time (see Section 3.1.1.2 about the requirements) and they allow easy identification of delivery threads in SMTPC. The type for this identifier is `sessta_id_T`.

To uniquely identify recipients they are enumerated within a transaction, so their format is `"%19s-%04X"` where the first 19 characters is the SMTPS transaction id and the last four characters are the recipient index (this limits the number of recipients per transaction to 65535 which seems to be enough[1]). The type for this identifier is `rcpt_id_T`.

[1]Famous last words? Probably need to increase this to 32 bits.

In some cases it is useful to have only one identifier in a structure and some field that denotes which identifier type is actually used, i.e., a superset of `sessta_id_T` and `rcpt_id_T`. This type is `smtp_id_T`.

Notice: under certain circumstances it might be worth to store identifiers as binary data instead of printable strings. For example, if they are used for large indices that are stored in main memory. For SMTP server identifiers this shrinks the size from 20 bytes down to 12 (8 byte for the counter, 4 for the process index assuming that it's a 32 bit system, i.e., the 8 bit process index will be stored in a 32 bit word). For the recipient identifiers the length will shrink from 25 bytes (probably stored in 28 bytes) down to 16 bytes. Whether it is worth to trade memory storage for conversion overhead (and added complexity) remains to be seen. In the first version of sendmail X this will probably not be implemented. See also Section 3.13.6.1.1 about size estimates.

4.1.2 Asynchronous Functions

This section talks about the integration of asynchronous functions with the event thread library. Section 3.1.1.1 gives a description of asynchronous functions, stating that a result (callback) function `RC` is invoked with the current status and the result of an aysnchronous function as parameters. However, the implementation of this is not as simple as it might seem. The event thread library (see Section 3.20.5 for a functional description and Section 4.3.8 for the implementation) uses a task context that is passed to worker threads which in turn execute the application function (passing it the entire task context).

4.1.2.1 Two Different Problems

Note: there are two different problems described here:

1. a caller may invoke an asynchronous function and the result of that function is the only result that is required.
2. a caller may invoke an asynchronous function and the result of that function is required by the caller for further operation.

Case 2 can be turned into case 1 by splitting a function into two ("continuation"), i.e., when the result is required to continue then the first half of the function saves its state and stops execution, e.g., it goes back to the wait queue. The second half of the function can then either be executed via the callback, or it is activated and combines the requested result with the current state to achieve the final result. Note: this kind of programming is not particularly simple as explained in Section 3.16.4, but it avoids blocking as much as possible.

4.1.2.2 Two Approaches

There are basically two approaches to this problem:

1. Use a callback function.
2. Add another queue (condqueue) to the event thread context and send a notification via the internal pipe that a task in that queue can be executed since the condition it is waiting for became true. A slight variation of this is to use a condition variable and another task that exclusively deals with this condition variable, but that just adds additional code. Unfortunately there is no (Unix) function that can wait for an I/O event and a condition variable.

The advantage of solution 1 is that it requires less context switching (even if it is only a thread context). If the function `RC` is a callback that is directly invoked from the function that receives the result from an external source, then the result values should not be those that are returned to the worker manager (e.g., OK, WAITQ, RUNQ, DEL, see 3.20.5.1.1), unless we put an extra handler inbetween that knows about these return values and can handle them properly by performing the appropriate actions. Therefore this approach requires careful programming, see below for details.

Solution 2 does not mess around with the worker function and the task (event thread) context without telling the event thread system about it, hence it does not have the programming restriction mentioned above; it allows for a more normal programming style. However, there needs to be a way to match the condition that became true with the task that is waiting for it. Usually this is done via identifiers (see Section 3.1.1.2), however, the current implementation only allows for a single character to be sent over the internal notification pipe[2]. Hence it is complicated to find the right task – waking up one after another to let them figure out themselves whether "their" condition is met can be expensive. One possible solution for this is to maintain a queue of results instead of writing the results directly into some structure. Elements in the result queue contain the necessary identifier to find the right task that is waiting for the condition/result.

An asynchronous call sequence looks like this:

1. Task `C` calls `create_request(request_data, context)` which appends `(request_data, context)` to a queue that is specific to the module that handles the request. `request_data` must contain a unique identifier (token) `T`, which is used to identify the original task (or at least the data structure which will store the result) that made a request when a result is returned (see Section 3.1.1.1). Task `C` usually returns `EVTHR_OK` to the worker thread, i.e., it is not appended to the wait queue, unless it is a task that can run concurrently and the context for the request is independent of the (event thread) task context. For example, QMGR sends requests to SMAR but it puts the task that sends those requests back into the wait queue since the result is handled by a different task (the one which communicates with SMAR). There is also no explicit context since the token `T` identifies the data structure in which the result will be stored later on (a recipient context in the active queue).

2. A function `request_handler()` takes a request out of its queue and performs the appropriate actions. Note: if "the appropriate actions" are fast, i.e., non-blocking, then the functionality of `request_handler()` can be integrated into `create_request()`. However, these asynchronous functions usually invoke some I/O operations which may block, i.e., even a send (write) operation may block if the internal (OS) buffers are filled. Hence a write task that is managed by the event thread system should be used. Moreover, if there is a single communication pipe to a server, then a single communication task should be used to avoid that requests are garbled (mixed with others) when being sent to the server (e.g., if two independent threads send request over one pipe the data might be interleaved and hence unusable).

3. The function `result_handler()` receives a result, possibly transforms the result into a format that is more suitable for the application, e.g., a package from a DNS server is translated into a C structure and looks up the identifier `T` in a table.

   The next step depends on which solution has been chosen:

   (a) For approach 1 the callback function is invoked with the return value, the token, and the context (which might be the task structure). The callback function performs some actions using the result value, e.g., store it in a data structure. Then it may inform some task about the fact that new data is available for processing, or it may put the requesting task back into the wait queue. Finally the callback function returns to `result_handler()`. The

[2] otherwise the write operation could theoretically block

`result_handler()` function may have placed itself back into the wait queue before invoking the callback function, and then simply returns to the worker manager.

If the task `C` is invoked via the callback from `result_handler()` then we either have to make `C` aware of this (implicitly by "knowing" which parts of `C` are invoked by callbacks or explicitly by passing this information somehow, e.g., encoded in the task structure). We can either require that a callback has different set of return values, i.e., only `EVTHR_OK`, or we need an "inbetween" function that can interpret the result values for a manager thread and manipulate the task context accordingly. The latter seems like the most generic and clean solution. Notice, however, that we have to take care of the usual problem with accessing the task while it is outside of our control, i.e., if it has been returned to the event thread system earlier on, the well-known problems arise (see Section 3.20.5.1.5, additionally the task must know whether it placed itself earlier on into the wait queue to avoid doing it again).

(b) For approach 2 the return value and the task structure (context) are appended to a queue, and a signal is somehow sent (e.g., condition variable or write on internal pipe), then the task waits for the next request, i.e., it returns to the worker manager.

4.1.2.3 Asynchronous Functions: Callback Synchronization

In the previous section solution 3a for approach 1 states that the callback function may store the result in a data structure. If the caller of the asynchronous function waits for the result, then the access to the structure must be synchronized and the caller must be informed that the data is available (valid). Note: this is *not* a good solution to the problem of asynchronous functions (because the caller blocks while waiting without giving the event threads library a chance to switch to another thread; if too many functions are doing this the system may even deadlock), but just a simple approach until a better solution is found.

The simple algorithm uses a mutex (to protect access to the shared data structure and the status variable), a condition variable (to signal completion), and a status variable which has three states:

1. initial state;
2. the caller is waiting for the result;
3. the result has been written.

Note: due to the asynchronous operation, states 2 and 3 do not need to happen in that order. Moreover, state 2 may not be reached at all because the callee is done before the caller needs the result (this is the best case because it avoids waiting).

Caller:

```
status = init;                  /* initialize system */
invoke callee                   /* call asynchronous function */
...                             /* do something */
lock;                           /* acquire mutex */
if (status == init) {           /* is it still the initial value? */
  status = wait;                /* indicate that caller is waiting */
  while (status == wait)        /* wait until status changes */
    cond_wait  /* wait for signal from callee, this also unlocks the mutex */
  }
}
unlock;
```

Callee:

```
...                                /* compute result */
lock;                              /* acquire mutex */
v = result;                        /* set result */
notify = (status == wait);         /* is caller waiting? */
status = sent;                     /* result is available */
if (notify)
  cond_signal                      /* notify caller if it is waiting */
unlock;                            /* done */
```

This can be extended if there are multiple function calls and hence multiple results to wait for. It would be ugly to use an individual status variable for each function, hence a status variable and a counter are used. If the counter is zero, then all results are available. The status variable simply indicates whether the caller is waiting for a result and hence the callee should use the condition variable to signal that the result is valid. Since there can be multiple callees, only the one for which the counter is zero and the status variable is `wait` must signal completion.

Caller:

```
status = init;                          /* initialize system */
counter = 0;
...
lock
++counter;
invoke callee                           /* call asynchronous function */
unlock                                  /* (maybe multiple times) */
...                                     /* do something */
lock                                    /* acquire mutex */
if (counter > 0) {                      /* are there outstanding results? */
  status = wait;                        /* indicate that caller is waiting */
  while (status == wait && counter > 0) /* wait for results */
    cond_wait                           /* wait for signal from callee */
}
unlock                                  /* done */
```

Callee:

```
...                                       /* compute result */
lock                                      /* acquire mutex */
--counter;
v = result;                               /* set result */
/* is caller waiting and this is the last result? */
if (status == wait && counter == 0) {
  status = sent;                          /* results are available */
  cond_signal                             /* notify caller */
}
unlock                                    /* done */
```

Notes:

- As usual a `while` loop is used around `cond_wait` to deal with spurious wakeups.
- It would be sufficient to use `(counter > 0)` as condition in the `while` loop of the caller.
- This algorithm allows the caller only to continue if all results are available. It can be useful to use a result value as soon as it is available. *Question*: how to implement this?

If the result value cannot have all possible values in its range, then two values can be designated as `v-init` and `v-wait` while all others can be considered as `v-sent`. This removes the need for a status variable because the result variable `v` itself is used for that purpose:

Caller:

```
v = v_init;                     /* initialize system */
invoke callee                   /* call asynchronous function */
...                             /* do something */
lock                            /* acquire mutex */
if (v == v_init) {              /* is it still the initial value? */
  v = v_wait;                   /* indicate that caller is waiting */
  while (v == v_wait)           /* wait until status changes */
    cond_wait                   /* wait for signal from callee */
  }
}
unlock
```

Callee:

```
...                             /* compute result */
lock;                           /* acquire mutex */
notify = (v == v_wait);         /* is caller waiting? */
v = result;                     /* set result */
if (notify)
  cond_signal                   /* notify caller if it is waiting */
unlock                          /* done */
```

4.1.2.4 Asynchronous Functions: Result Queue

Solution 2 described in Section 4.1.2.2 needs some form of synchronization too: as described in the previous Section 4.1.2.3 the callee may may return a result before or after the caller wants to access it.

Using the first algorithm from the previous section with a slight modification like this:

Caller:

```
status = init;                  /* initialize system */
invoke callee                   /* call asynchronous function */
...                             /* do something */
lock;                           /* acquire mutex */
waiting = (status == init);     /* is it still the initial value? */
if (waiting)
  status = wait;                /* indicate that caller is waiting */
```

```
unlock;
if (waiting)
  put task into condqueue
```

Callee:

```
...                             /* compute result */
lock;                           /* acquire mutex */
v = result;                     /* set result */
notify = (status == wait);      /* is caller waiting? */
status = sent;                  /* result is available */
if (notify)                     /* if caller is waiting: */
  cond_signal                   /* notify evthr system about condition */
unlock;                         /* done */
```

causes a race condition: after unlocking the mutex in the caller but before putting the task into condqueue the callee may signal the event thread library that condition is fulfilled. To avoid this race condition, a condition variable and a mutex must be combined as it is done by `pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`:

> The `pthread_cond_wait(3)` function atomically blocks the current thread waiting on the condition variable specified by cond, and unblocks the mutex specified by mutex.

In the modified version the caller just instructs the event thread library to put the task into the condqueue:

```
status = init;                  /* initialize system */
invoke callee                   /* call asynchronous function */
...                             /* do something */
lock;                           /* acquire mutex */
waiting = (status == init);     /* is it still the initial value? */
if (waiting) {
  status = wait;                /* indicate that caller is waiting */
  return to event thread library
  with return code that requests to
  put task into condqueue
}
```

and the library itself unlocks the mutex afterwards. A condition variable for event threads hence consists of a mutex and an identifier (see Section 3.1.1.2) to match caller and callee. Such a condition variable belongs to a task (just like a file descriptor on which a task might be waiting).

4.1.3 Transaction Based Processing

To deal with various kinds of errors, it is necessary to write functions such that they are "reversible", i.e., they either perform a transaction or they don't change the state (in an inconsistent manner). Unfortunately, it is fairly complicated to write all functions in such a way. For example, if multiple changes must be made each of which can fail, then either the previous state must be preserved such that it can be restored when something goes wrong, or the changes must be undone individually. For this to work properly, it is essential that the "undo" operation itself can not fail. Hence an "undo" operation must not rely on resources that might become unavailable during processing; for example, if it requires memory, then that memory must be allocated before the whole operation is started, or at least while the invidual changes are made, such that the "undo" operation does not need to allocate memory. This might not be achievable in some cases, however. For example, there are operations in sendmail X that require updates to two persistent databases, i.e., DEFEDB and IBDB. Both of these require disk space which cannot be pre-allocated nor can it be guaranteed that two independent disk write operations will succeed. If the first one fails, then we don't need to perform the second one. However, if the first one succeeds and the second one fails, then there is no guarantee that the first operation can be undone. Hence the operations must be performed in such a manner that even in the worst case mail will not be lost but at most delivered more than once.

Transaction based processing is also fairly complicated in case of asynchronous operations. If a function has to perform changes to a local state and creates a list of change requests which will be performed later on by some asynchronously running thread, then that list of change requests should offer a callback functionality which can be invoked with the status of the operation such that it can either perform the local changes (if that has not been done yet) or undo the local changes (if they have been performed earlier). An additional problem is that other functions may have performed changes inbetween. Hence it is not possible to save the old state and restore it in case of an error because that would erase other (valid) changes. Instead "relative" changes must be performed, e.g., in/decreasing a counter.

4.1.4 Secure Programming

As mentioned in Section 2.14, item 8, C has some inherent security problems because it does not provide boundary checks on buffers, arrays, or strings. Those checks must be implemented by the program itself. Here are some (obvious) hints when writing code for sendmail X (or other secure programs):

- Functions that do not check the boundaries of output data *must not* be used, e.g., `gets(3)`, `strcpy(3)`, `strcat(3)`, etc.

The sendmail X libraries provide communication buffers and replacements for strings, see 3.16.11.1.1, and 3.16.7, which must be used as much as possible; exception should only be made for external libraries that do not support these data structures.

4.2 Schedule

Order of implementation:

1. Basic foundation (Milestone 0)
   - Learn autoconf, automake, etc use it.
   - Error handling routines
   - Memory allocation
   - I/O ?
   - Threading library (libsmi?)
2. SMTP Server Prototype (Milestone 1)
   based on state threads

3. QMGR Prototype (Milestone 2)
   - based on POSIX threads and a worker thread library.
   - implement communication framework
   - implement communication between QMGR and SMTPS
4. SMTP Client Prototype (Milestone 3)
   - based on state threads
   - implement communication between QMGR and SMTPC

   Milestone 3 is complete if a mail can be received by SMTPS, safely stored in a queue by QMGR, scheduled for delivery by a very simple scheduler in the QMGR, and then delivered by SMTPC.

   Milestone 3 has been reached 2002-08-19.

   Notice: this is prototype code. There are many unfinished places within the existing code not to mention all the parts that are completely missing.
5. AR Prototype (Milestone 4) (called SMAR: SendMail Address Resolver)
   - based on event thread library.
   - implement communication between QMGR and AR, including proper handling of (unresolved) addresses.
   - implement a very simple routing feature that maps domain parts to IPv4 addresses.
     Milestone 4 has been reached 2002-09-19.
6. Implement deferred queue (done, uses Berkeley DB 4.1.x).
7. Implement a simple anti-relay system (done, using regular expressions).
8. Routing MTA (with minimal functionality). Milestone 5 has been reached.
9. MCP to control system. Milestone 6 (implemented).
10. implement more data structures in the QMGR, esp. the connection caches, to provide better scheduling (done).
11. configuration system (done).

Next steps:

- Basic work:
  - logging (some work has been done already).
- hard task: a *real* scheduler.

As of 2004-01-01 sendmail X is running as MTA on the machine of the author. sendmail 8 is only used for mail submission.

4.3 Libraries

This section describes the implementation of some libraries which are used by various sendmail X modules.

4.3.1 Queues, Lists, etc

Queues, lists, et.al., are taken from OpenBSD `<sys/queue.h>`.

4.3.2 Hash Tables

Two hash table implementations are available: a conventional one with one key and a version with two keys. The latter is currently not used, it was intended for the DA database.

4.3.3 Classes

Classes can be used to check whether a key is in a set.

Classes can be implemented via a hash table: sendmail 8 allows up to 256 classes (there is a simple mapping of class names to single byte values). Each element in a class is stored in a hash table, the RHS is a bitmap to indicate to which classes it belongs. Checking whether something is in a class is done as follows: lookup word in hash table: if it exists: is it a member of the class we are looking for (which is just a `bitnset()` test)? This kind of implementation uses one hash table for many classes.

Alternative implementations are lists if a class has only a few elements (linear search) or tree.

4.3.4 Trees

Various versions of balanced trees are available which are based on code that has been found on the internet. See `include/sm/tree.h`, `include/sm/avl.h`, and `include/sm/bsd-tree.h`. The latter is taken from OpenBSD `<sys/tree.h>`, it can be used to generate functions that operate on splay trees and red-black trees. That is, there aren't generic functions that operate on trees, but specific functions are generated that operate on one type. Of course it is possible to generate a generic version where a tree node contains the usual data, i.e., a key and a pointer to the value.

4.3.5 RSC

A restricted size cache is implemented using a hash table (for access) and a linked list which is kept in most-recently used order (MRU).

Note: if we don't need to dynamically delete and add entries from the RSC and we don't need the MRU feature, then we can probably use a (binary) hash table which keeps tracks of the number of entries in it.

The structure definitions given below are hidden from the application.

```
typedef struct rsc_S rsc_T, *rsc_P;

struct rsc_entry_S
{
  CIRCLEQ_ENTRY(rsc_entry_S)       rsce_link;     /* MRU linkage */
  const char                      *rsce_key;      /* lookup key */
  unsigned int                     rsce_len;      /* key len */
  void                            *rsce_value;    /* corresponding value */
};
```

```
struct rsc_S
{
  bht_P                          rsc_table;  /* table with key, rsc_entry pairs */
  unsigned int                   rsc_limit;  /* max # of entries */
  unsigned int                   rsc_used;   /* current # of entries */
  rsc_create_F                   rsc_create; /* constructor */
  rsc_delete_F                   rsc_delete; /* destructor */
  CIRCLEQ_HEAD(, rsc_entry_S)    rsc_link;   /* MRU linkage */
  void                          *rsc_ctx;    /* application context */
};
```

The following functions (and function types) are provided:

```
typedef void    *(*rsc_create_F) (const char *_key, unsigned int _len,
                                  void *_value, void *_ctx);
typedef sm_ret_T (*rsc_delete_F) (void *_value, void *_ctx);
typedef void (rsc_walk_F)(const char *_key, const void *_value, const void *_ctx);

extern rsc_P       rsc_create(sm_rpool_P _rpool, unsigned int _limit, unsigned int _htsize,
                              rsc_create_F _create, rsc_delete_F _delete, void *_ctx);
extern void        rsc_free(rsc_P _rsc);
extern void        rsc_walk(rsc_P _rsc, rsc_walk_F *_f);
extern sm_ret_T    rsc_add(rsc_P _rsc, bool _delok, const char *_key,
                           unsigned int _len, void *_value, void **_pvalue);
extern const void *rsc_lookup(rsc_P _cache, const char *_key, unsigned int _len);
extern sm_ret_T    rsc_rm(rsc_P _rsc, const char *_key, unsigned int _len);
extern int         rsc_usage(rsc_P _rsc);
extern void        rsc_stats(rsc_P _rsc, char *_out, unsigned int _len);
```

4.3.5.1 Typed RSC

If an RSC stores data of different types, it must be possible to distinguish between them. This is necessary for functions that "walk" through an RSC and perform operations on the data or just for generic functions, i.e., create and delete. As such, the RSC could store an additional type and pass it to the delete and create functions. However, this could also handled by the application itself, i.e., it adds a unique identifier to the data that it stores in an RSC.

The API of a typed RSC implementation differs as follow:

- `struct rsc_entry_S` has an addition element rsce_type.
- The function prototypes `rsc_create_F`, `rsc_delete_F`, and `rsc_walk_F` have an additional parameter `type`.
- `rsc_add` also has an additional parameter `type`.

Note that this causes problems with a unified API as described in Section 3.16.12.

4.3.6 DBs with Multiple Access Keys

We could use a fixed number of keys and set those that we are not interested in to NULL, or we could use a variable number of keys and specify a key and an index. How easy is the second solution to implement? The keys might be implemented as an array with some upper limit that is specified at creation time.

A possible approach to allow for any number of keys is to specify an array of keys (and corresponding lengths). During initialization the maximum number of keys is defined (and stored in the structure describing the DB). Access methods pass in a key and length and the index of the key, where zero is the primary key. When an element is entered, the primary key is used. What about the other keys? They must be specified too since the appropriate entries in the hash tables must be constructed. This seems to become ugly. Let's try a simpler approach first (one, two, three keys).

4.3.7 DBs with Non-unique Access Keys

This might be implemented by having a (second) link in the entries which points to the next element with the same key. Check the hash table implementation.

4.3.8 Event Thread Library

The event thread library provides the basic framework for a worker based thread pool that is driven by I/O events and by wakeup times for tasks. See Section 3.20.5 for a functional description.

This library uses a general context that describes one event thread system and a per task context. It uses two queues: a run queue of tasks that can be executed, and a wait queue of tasks that wait for some event. Each task is in exactly one of the queues at each time.

The event thread context looks like this (2004-07-27):

```
struct sm_evthr_ctx_S
{
        pthread_cond_t   evthrc_cv;
        pthread_mutex_t  evthrc_waitqmut;
        pthread_mutex_t  evthrc_runqmut;

        int              evthrc_max;    /* max. number of threads */
        int              evthrc_min;    /* min. number of threads */
        int              evthrc_cur;    /* current number of threads */
        int              evthrc_idl;    /* idle threads */
        int              evthrc_stop;   /* stop threads */
        int              evthrc_maxfd;  /* maximum number of FDs */
        timeval_T        evthrc_time;   /* current time */
        sm_evthr_task_P *evthrc_fd2t;   /* array to map FDs to tasks */

        /* pipe between control thread and worker/signal threads */
        int              evthrc_pipe[2];

        CIRCLEQ_HEAD(, sm_evthr_task_S) evthrc_waitq;
        CIRCLEQ_HEAD(, sm_evthr_task_S) evthrc_runq;
};
```

The system maintains some number of worker threads between the specified minimum and maximum. Idle threads vanish after some timeout until the minimum number is reached.

An application function has this prototype:

```
typedef sm_ret_T (evthr_task_F)(sm_evthr_task_P);
```

That is, the function receives the per-task context, whose structure is listed next, as a parameter. Even though this is a violation of the abstraction principle, it allows for some functionality which would be awkward to achieve otherwise. For example, a user application can directly manipulate the next wakeup time. We could hide this behind a void pointer and provide some functions to manipulate the task context if we want to hide the implementation.

```
struct sm_evthr_task_S
{
        CIRCLEQ_ENTRY(sm_evthr_task_S)  evthrt_next;
        /* protects evthrt_rqevf and evthrt_sleep */
        pthread_mutex_t  evthrt_mutex;
        int              evthrt_rqevf;  /* requested event flags; see below */
        int              evthrt_evocc;  /* events occurred; see below */
        int              evthrt_state;  /* current state; see below  */
        int              evthrt_fd;     /* fd to watch */
        timeval_T        evthrt_sleep;  /* when to wake up */
        evthr_task_F    *evthrt_fct;    /* function to execute */
        void            *evthrt_actx;   /* application context */
        sm_evthr_ctx_P   evthrt_ctx;    /* evthr context */
        sm_evthr_nc_P    evthrt_nc;     /* network connection */
};
```

The first element describes the linked list, i.e., either the wait queue or the run queue. The second element (requested event flags) lists the events the task is waiting for:

flag	meaning
EVT_EV_RD	task is waiting for fd to become ready for read
EVT_EV_WR	task is waiting for fd to become ready for write
EVT_EV_LI	task is waiting for fd to become ready for accept
EVT_EV_SL	task is waiting for timeout

The third element (events occurred) lists the events that activated the task:

flag	meaning
EVT_EV_RD_Y	task is ready for read
EVT_EV_WR_Y	task is ready for write
EVT_EV_LI_Y	task received a new connection
EVT_EV_SL_Y	task has been woken up (after timeout)

The fourth element (status) consists of some internal state flags, e.g., in which queue the task is and what it wants to do next.

The event thread library provides these functions:

```
extern sm_ret_T  evthr_init(sm_evthr_ctx_P *_pctx, int _minthr, int _maxthr, int _maxfd);
```

```
extern sm_ret_T  evthr_task_new(sm_evthr_ctx_P   _ctx,
                        sm_evthr_task_P        *_task,
                        int                     _ev,
                        int                     _fd,
                        timeval_T              *_sleept,
                        evthr_task_F           *_fct,
                        void                   *_taskctx);
extern sm_ret_T  evthr_loop(sm_evthr_ctx_P _ctx);
extern sm_ret_T  evthr_waitq_app(sm_evthr_task_P _task);
extern sm_ret_T  evthr_en_wr(sm_evthr_task_P _task);
extern sm_ret_T  evthr_time(sm_evthr_ctx_P _ctx, timeval_T *_ct);
extern sm_ret_T  evthr_new_sl(sm_evthr_task_P _task, timeval_T _slpt, bool _change);
```

An application first initializes the library (`evthr_init()`), then it creates at least one task (`evthr_task_new()`), and thereafter calls `evthr_loop()` which in turn monitors the desired events (I/O, timeout) and invokes the callback functions. Those callback functions return a result which indicates what should happen with the task:

flag	meaning
EVTHR_OK	do nothing, task has been taken care of
EVTHR_WAITQ	put in wait queue
EVTHR_RUNQ	put in run queue
EVTHR_SLPQ	sleep for a while
EVTHR_DEL	delete task
EVTHR_TERM	terminate event thread loop

Section 3.20.5 describes some problems that need to be solved:

1. Put a task back into the wait queue, this can be done via evthr_waitq_app().
2. Enable the event type EVT_EV_WR, this can be done via evthr_en_wr().
3. Change (shorten) the timeout, this can be done via evthr_new_sl().

Question: what about callback functions for signals? Currently the usual signals terminate the system.

4.3.9 RCB Communication

sendmail X uses RCBs for communication between the various modules (see Section 3.16.11.1.1). This section describes some functions that simplify handling of RCB based communication for modules which use the event thread library (see Section 4.3.8).

This library uses the following structure:

```
struct rcbcom_ctx_S
{
  sm_evthr_task_P             rcbcom_tsk;    /* event thread task */
  sm_rcb_P                    rcbcom_rdrcb;  /* RCB for communication (rd) */
  SIMPLEQ_HEAD(, sm_rcbe_S)   rcbcom_wrrcbl; /* RCB list (wr) */
  pthread_mutex_t             rcbcom_wrmutex;
};
```

The first entry is a pointer to the task that is responsible for the communication. The second entry is the RCB in which data is received. The third is a list of RCBs for data that must be send out, this list is proteced by the mutex that is the last element of the structure.

The following functions are provided by the library:

`sm_ret_T sm_rcbcom_open(rcbcom_ctx_P rcbcom_ctx)`: create a RCB communication context.

`sm_ret_T sm_rcbcom_close(rcbcom_ctx_P rcbcom_ctx)`: close a RCB communication context.

`sm_ret_T sm_rcbe_new_enc(sm_rcbe_P *prcbe, int minsz)`: create an entry for the write RCB list and open it for encoding.

`sm_rcbcom_prerep(rcbcom_ctx_P rcbcom_ctx, sm_evthr_task_P tsk, sm_rcbe_P *prcbe)`: prepare a reply to a module: close the read RCB after decoding, open it for encoding, create a new RCB for writing (using `sm_rcbe_new_enc()`), and put the task back into the wait queue (unless `tsk` is `NULL`). Notice: after calling this function (with `tsk` not `NULL`), the task is not under control of the caller anymore, hence all manipulation of its state must be done via the functions provided by the event thread library.

`sm_rcbcom_endrep(rcbcom_ctx_P rcbcom_ctx, sm_evthr_task_P tsk, bool notified, sm_rcbe_P rcbe)`: close the write RCB, append it to the RCB list in the context, and if `notified` is not set inform the task about the new write request (using `evthr_en_wr()`).

`sm_rcbcom2mod(sm_evthr_task_P tsk, rcbcom_ctx_P rcbcom_ctx)`: send the first element of the RCB write list to filedescriptor specified in the task. This function should be called when the event threads library invoke the callback for the task and denotes that the file descriptor is ready for writing, i.e., it will be called from an event thread task that checks for I/O activity on that file descriptor. Since the function does not conform to the function specification for a task, it is called by a wrapper that extracts the RCB communication context from the task (probably indirectly). The function will only send the first element of the list; if the list is empty afterwards, it will disable the write request for this task, otherwise the event thread library will invoke the callback again when the filedescriptor is ready for writing. This offers a chance for an optimization: check whether another RCB can be written; question: is there a call that can determine the free buffer size for the file descriptor such that the next write operation does not block?

4.3.10 DNS

An asynchronous DNS resolver has been implemented (`libdns/`) according to the specification of Section 3.16.13.

4.3.10.1 DNS Data Structures

A DNS request has the following structure:

```
struct dns_req_S
{
        sm_cstr_P        dnsreq_query;  /* the query itself */
        time_T           dnsreq_start;  /* request start time */
/*      unsigned int     dnsreq_tries;   * retries */
        dns_type_T       dnsreq_type;   /* query type */
        unsigned short   dnsreq_flags;  /* currently: is in list? */
        sm_str_P         dnsreq_key;    /* key for hash table: query + type */
```

```
        dns_callback_F  *dnsreq_fct;    /* callback into application */
        void            *dnsreq_ctx;    /* context for application callback */
        TAILQ_ENTRY(dns_req_S)  dnsreq_link;    /* next entry */
};
```

A DNS result consists of a list of entries with the following elements:

```
typedef union
{
        ipv4_T           dnsresu_a;
        sm_cstr_P        dnsresu_name;  /* name from DNS */
} dns_resu_T;

struct dns_res_S
{
        sm_cstr_P        dnsres_query;          /* original query */
        sm_ret_T         dnsres_ret;            /* error code */
        dns_type_T       dnsres_qtype;          /* original query type */
        unsigned int     dnsres_entries;        /* number of entries */
        unsigned int     dnsres_maxentries;     /* max. number of entries */
        dns_resl_T       dnsres_hd;             /* head of list of mx entries */
};

struct dns_rese_S
{
        dns_type_T      dnsrese_type;   /* result type */
        unsigned int    dnsrese_ttl;    /* TTL from DNS */
        unsigned short  dnsrese_pref;   /* preference from DNS */
        unsigned short  dnsrese_weight; /* for internal randomization */
        sm_cstr_P       dnsrese_name;   /* RR name */
        TAILQ_ENTRY(dns_rese_S) dnsrese_link;   /* next entry */
        dns_resu_T      dnsrese_val;
};
```

A DNS manager context contains the following elements:

```
struct dns_mgr_ctx_S
{
        unsigned int     dnsmgr_flags;
        dns_rql_T        dnsmgr_req_hd;    /* list of requests */
        dns_req_P        dnsmgr_req_cur;   /* current request */

        /* hash table to store requests */
        bht_P            dnsmgr_req_ht;

#if SM_USE_PTHREADS
        pthread_mutex_t  dnsmgr_mutex;  /* for the entire context? */
        sm_evthr_task_P  dnsmgr_tsk;    /* XXX Just one? */
        sm_evthr_task_P  dnsmgr_cleanup;
#endif /* SM_USE_PTHREADS */
};
```

and a DNS task looks like this:

```
struct dns_tsk_S
{
        /* XXX int or statethreads socket */
        int             dnstsk_fd;            /* socket */
        dns_mgr_ctx_P   dnstsk_mgr;           /* DNS manager */
        uint            dnstsk_flags;         /* operating flags */
        uint            dnstsk_timeouts;      /* queries that timed out */
        sockaddr_in_T   dnstsk_sin;           /* socket description */
        sm_str_P        dnstsk_rd;      /* read buffer */
        sm_str_P        dnstsk_wr;      /* write buffer */
};
```

4.3.10.2 DNS Functions

The DNS library offers functions to create and delete a DNS manager context:

```
sm_ret_T  dns_mgr_ctx_new(uint flags, dns_mgr_ctx_P *pdns_mgr_ctx);
sm_ret_T  dns_mgr_ctx_del(dns_mgr_ctx_P dns_mgr_ctx);
```

and similar functions for DNS tasks:

```
sm_ret_T  dns_tsk_new(dns_mgr_ctx_P dns_mgr_ctx, uint flags, ipv4_T ipv4,
                      dns_tsk_P *pdns_tsk);
sm_ret_T  dns_tsk_del(dns_tsk_P dns_tsk);
sm_ret_T  dns_tsk_start(dns_mgr_ctx_P dns_mgr_ctx, dns_tsk_P dns_tsk,
                        sm_evthr_ctx_P evthr_ctx)
```

An application can make a DNS request using the following function:

```
sm_ret_T  dns_req_add(dns_mgr_ctx_P dns_mgr_ctx, sm_cstr_P query,
                      dns_type_T type, dns_callback_F *fct, void *ctx);
```

Internal functions of the library are:

```
sm_ret_T  dns_comm_tsk(sm_evthr_task_P tsk);
sm_ret_T  dns_tsk_cleanup(sm_evthr_task_P tsk);
sm_ret_T  dns_tsk_rd(sm_evthr_task_P tsk)
sm_ret_T  dns_tsk_wr(sm_evthr_task_P tsk)
void      dns_req_del(void *value, void *ctx);
sm_ret_T  dns_receive(dns_tsk_P dns_tsk);
sm_ret_T  dns_send(dns_tsk_P dns_tsk);
sm_ret_T  dns_decode(sm_str_P ans, uchar *query, int qlen, dns_type_T *ptype,
                     dns_res_P dns_res);
```

`dns_comm_tsk()` is a function that is called whenever I/O activity is possible, it invokes the read and write functions `dns_tsk_rd()` and `dns_tsk_wr()`; respectively. `dns_tsk_cleanup()` is cleanup task that deals with requests which didn't receive an answer within a certain amount of time.

4.3.10.2.1 Initializing and Starting the DNS Resolver The following steps are necessary for initializing and starting the DNS resolver (when using the event threads system):

```
/* Initialize DNS resolver */
ret = dns_rslv_new(random);

/* Create DNS manager context */
ret = dns_mgr_ctx_new(0, &dns_mgr_ctx);

/* Create one DNS resolver task */
ret = dns_tsk_new(dns_mgr_ctx, 0, ipv4, &dns_tsk);

/* Start DNS tasks (resolver and cleanup) */
ret = dns_tsk_start(dns_mgr_ctx, dns_tsk, evthr_ctx);
```

4.3.10.2.2 Making a DNS Query `dns_req_add()` creates a DNS request and adds it to the list maintained in the DNS manager context (`dnsmgr_req_hd`), unless such a request is already queued, which is checked via the hash table `dnsmgr_req_ht`, to which the request is always added. The key for the hash table is the DNS query string and the DNS query type concatenated (if the query string has a trailing dot it will be removed). Whenever a DNS request is added to the list the write event is triggered for `dns_tsk_wr()`, which will take the current request (pointed to by `dnsmgr_req_cur`), form a DNS query and send it to a DNS server using `dns_send()`. Notice: the request will not be removed from the list, this is done by either the cleanup task or the read function. To indicate whether a request is in the list or merely in the hash table, the field `dnsreq_flags` is used.

The hash table is used by the function `dns_tsk_rd()` which receives replies from a DNS server to identify all requests for a DNS query (it uses `dns_tsk_rd()` and `dns_decode()`). The function removes all those requests from the hash table and the DNS manager list, and creates a local list out of them. Then it walks through that list and invokes the callback functions specified in the requests with DNS result and the application specific context.

The cleanup task `dns_tsk_cleanup()` uses the DNS manager list of requests (from the first element up to the current one), and checks whether they have timed out (based on the time the request has been made `dnsreq_start`). If this is the case then the request is removed from the list and appended to a local list. Moreover, all requests in the hash table for the same query are moved into the local list too – this needs improvement. As soon as a request is found that is not timed out (or the current element is reached), the search stops (this currently doesn't allow for individual timeouts). Thereafter, the callbacks for the requests in the local lists are invoked with a DNS result that contains an error code (timeout).

4.3.11 Logging

There is currently a simple implementation for logging in `libmta/log.c`. It closely follows the ISC logging module API mentioned in Section 3.16.16.1.

4.3.11.1 Logfile Rotation

The current implementation has a slight problem with logfile rotation. The programs use stdout/stderr for logging; they do not open a named logfile themselves, this is done by the MCP for them. Hence there is no way for them to reopen a logfile, instead the `sm_log_reopen()` function rewinds the file using `ftruncate(2)` and `fseek(3)`.

Alternatively the name of a logfile can be passed to each module such that it can open the file itself and hence the filename is available for the reopen function. The filename could be stored in the logging context and hence the reopen function could act accordingly, i.e., use `sm_io_open()` if a filename exists and otherwise the method described above.

4.4 SMTP Server Daemon

4.4.1 First Prototype

The first prototype of SMTPS is based on state threads (see Section 3.20.3.1).

This prototype uses a set of threads which is limited by an upper and lower number. If threads are idle for some time and there are more than a specified number of idle threads available, they terminate. If not enough threads are available, new ones are created up to a specified maximum.

4.4.1.1 Misc

Remark (placed here so it doesn't get lost): there is a restricted number (< 60000) of possible open connections to one port. Could that limit the throughput we are trying to achieve or is such a high number of connections unfeasible?

4.4.2 Communication between SMTPS and QMGR

We do not want a separate connection between QMGR and SMTPS for each thread in SMTPS, hence we need to associate the data from QMGR with the right thread in SMTPS. One approach is to have a receiver thread in SMTPS which communicates with the QMGR. It receives all data from the QMGR and identifies the context (session/transaction) to which the data belongs. This needs some list of all contexts, e.g., an AVL tree, or, if the number of entries is small enough, a linear list. Question: is there some better method, i.e., instead of searching some structure have direct access to the right thread (see also Section 3.1.1.2)? There might be some optimization possible since each SMTPS has only a limited number of threads, so we could have an array of that size and encode an index into that array into the RCB, e.g., use another ID type that is passed around (like a context pointer). It then adds that data to the context and notifies the thread There is one thread to read from the communication channel, but multiple tasks can actually write to it; writing is sequentialized by a mutex. In a conventional thread model, we would just select on I/O activities for that channel and notifications when a new RCB is added to the list to send to the QMGR (like it is done in the QMGR), however, in state threads I/O activity is controlled by the internal scheduler. Since there are multiple threads, it might be necessary to control how far ahead the writers can be of the readers (to avoid starvation and unfairness). However, this should be self-adjusting since threads are waiting for replies for requests they send before they send out new ones (by default, in some cases a few requests may be outstanding from one thread). If too many threads send data, then the capacity of the communication channel and the way requests are handled by the QMGR should avoid starvation and guarantee fairness.

4.4.2.1 Implementation of Communication between SMTPS and QMGR

As explained above there is one thread that takes care of the communication between the module and the QMGR. This thread uses the following structure as context:

```
struct s2q_ctx_S
{
        int             s2q_status;    /* status */
        st_netfd_t      s2q_fd;        /* fd for communication */
        int             s2q_smtps_id;  /* smtps id */
        st_mutex_t      s2q_wr_mutex;  /* mutex for write */
        unsigned int    s2q_maxrcbs;   /* max. # of outstanding requests */
        unsigned int    s2q_currcbs;   /* current # of outstanding requests */
        sessta_id_P    *s2q_sids;      /* array of session ids */
        smtps_sess_P   *s2q_sess;      /* array of session ctx */
};
```

For initialization and termination of the communication task the following two functions are provided:

```
sm_ret_T sm_s2q_init(s2q_ctx_P s2q_ctx, int smtps_id, unsigned int maxrcbs);
sm_ret_T sm_s2q_stop(s2q_ctx_P s2q_ctx);
```

The initialization function connects to the QMGR and stores the file descriptor for communication in `s2q_fd`. It allocates two arrays for sessions IDs and session contexts which are used to find the SMTPS session for an incoming RCB, and it sends the initial "A new SMTPS has been started" to the QMGR. Finally `sm_s2q_init()` starts a thread that executes the function `void *sm_rcb_from_srv(void *arg)` which receives the s2q context as parameter. This function receives an RCB from QMGR and notifies the thread that is associated with the task via a condition variable; the thread can be found using the `s2q_sids` array.

Data can be sent to the QMGR using one of the functions `sm_s2q_*()` for new session ID, close session ID, new transaction ID, new recipient, close transaction ID, and discard transaction ID.

The function `sm_w4q2s_reply()` is used to wait for a reply from QMGR. It waits on a condition variable (which is stored in the SMTPS session context) which is signalled by `sm_rcb_from_qmgr()`.

4.4.2.2 SMTPS - QMGR Protocol

Initially the SMTP server sends the QMGR its id and the maximum number of threads it is going to create.

RT_S2Q_NID	id of new SMTPS
RT_S2Q_ID	id of SMTPS
RT_S2Q_CID	close SMTPS (id)
RT_S2Q_STAT	status
RT_S2Q_MAXTHRDS	max number of threads
RT_S2Q_NSEID	new session id
RT_S2Q_SEID	session id
RT_S2Q_CSEID	close session id
RT_S2Q_CLTIP4	client IPv4 address
RT_S2Q_CLTIP6	client IPv6 address
RT_S2Q_CLTPORT	client port
RT_S2Q_NTAID	new transaction id
RT_S2Q_TAID	transaction id
RT_S2Q_CTAID	close transaction id
RT_S2Q_DTAID	discard transaction id
RT_S2Q_MAIL	mail from
RT_S2Q_RCPT_IDX	rcpt idx
RT_S2Q_RCPT	rcpt to
RT_S2Q_CDBID	cdb id

The common reply format from QMGR to SMTPS consists of the SMTPS id (which is only transmitted for paranoia), a session or transaction id, a status code and an optional status text:

RT_Q2S_ID	SMTPS id
RT_Q2S_SEID/RT_Q2S_TAID	session/transaction id
RT_Q2S_STAT[D]	status (ok, reject, more detailed?)
RT_Q2S_STATT	status text

The function `sm_rcb_from_srv()` uses the session/transaction id to find the correct thread to which the rest of the RCB will be given.

RT_Q2S_ID	id of SMTPS
RT_Q2S_STAT	status for session/transaction/...
RT_Q2S_STATV	status value (text follows)
RT_Q2S_STATT	status text
RT_Q2S_SEID	session id
RT_Q2S_TAID	transaction id
RT_Q2S_RCPT_IDX	rcpt idx
RT_Q2S_CDBID	cdb id
RT_Q2S_THRDS	slow down
RT_Q2S_STOP	stop reception (use slow = 0?)
RT_Q2S_DOWN	shut down

4.4.2.3 SMTPS - SMAR Protocol

The SMTP server uses the AR as map lookup server to avoid blocking calls in the state-threads application. While the anti-spam logic etc is implemented in SMTPS, the map lookups are performed by SMAR. Hence SMTPS only sends minimal information to SMAR, e.g., the sender or recipient address and asks for lookups in some maps with certain features, e.g., lookup the full address, the domain part, the address without details ("+detail").

RT_S2A_TAID	transaction id
RT_S2A_MAIL	mail from
RT_S2A_RCPT_IDX	rcpt idx
RT_S2A_RCPT	rcpt to (printable address)
RT_S2A_LTYPE	lookup type
RT_S2A_LFLAGS	lookup flags

To simplify the SMTP server code, the reply format for SMAR is basically the same as for QMGR:

RT_A2S_ID	id of SMTPS
RT_A2S_TAID	transaction id
RT_A2S_STAT[D]	status (ok, reject, more detailed?)
RT_A2S_STATT	status message

RT_A2S_MAIL_ST	mail status
RT_A2S_RCPT_IDX	rcpt index
RT_A2S_RCPT_ST	rcpt status

Values for lookup types are:

LT_LOCAL_ADDR	is local address?
LT_RCPT_OK	is address ok as a recipient?
LT_MAIL_OK	is address ok as a sender?

Values for lookup flags are:

LF_DOMAIN	try domain
LF_FULL	try full address
LF_LOCAL	try localpart
LF_NODETAIL	try without detail

4.5 Mail Delivery Agents

4.5.1 Maintaining Delivery Classes and Agents

As explained elsewhere (2.8.2) it is possible to specify multiple delivery classes and multiple delivery agents that implement delivery classes. The former are referenced by the address resolver when selecting a *mailer*. The latter are selected by the scheduler after it receives a *mailer* to use.

Every delivery agent has an index and a list of delivery classes it implements. There is also a list of delivery classes (which are referenced by some id, most likely a numeric index into an array). This list is maintained by SMAR, each DA, and QMGR (and must obviously be kept in sync if numeric indices are used instead of names). QMGR keeps for each delivery class a list of delivery agents that implement the class, which can be used by the scheduler to select a DA that will perform a delivery attempt.

Note: As described in Section 3.8.3.1 item 6, the first version of sendmail X does not need to implement the full set of this; all delivery agents implement the same delivery classes, hence they can be selected freely without any restriction.

4.6 SMTP Client

4.6.1 First Prototype

The first prototype of SMTPC is based on state threads (see Section 3.20.3.1).

It follows a similar thread model as that used for the SMTP server daemon (see Section 4.4).

4.6.1.1 SMTPC - QMGR Protocol

See Section 3.4.5.1 for a description.

As usual, a protocol header is sent first. Moreover, the next entry in each RCB is the identifier of the SMTPC to which the QMGR wants to talk: RT_Q2C_ID.

The rest of the RCB is described below for each function.

Notice: for status codes an additional text field might follow, which currently isn't specified here.

1. qmgr_da_new(IN da, IN da-descripton, OUT da-handle, OUT status)
2. qmgr_da_new_result(IN da, IN da-handle, OUT status)
3. qmgr_da_stop(IN da, IN da-handle, OUT status)
4. qmgr_da_stop_result(IN da, IN da-handle, OUT status)
5. qmgr_da_sess_new(IN da, IN session, IN da-session-handle, IN transaction, IN da-trans-handle, OUT session-status, OUT trans-status, OUT status)

   RT_Q2C_DCID: delivery class id.

   - RT_Q2C_NSEID: open a new session (using this id), or alternatively: RT_Q2C_ONESEID: session with one transaction: this sends just the data for a session with a single transaction (included in this RCB), qmgr_da_session_close() will not be called afterwards.
   - RT_Q2C_SRVIP4: server IPv4 address, or RT_Q2C_SRVIP6: server IPv6 address; optional: port number (defaults to 25).

   More data to follow, e.g., requirements about the session.

   For the transaction data see below (item 7).
6. qmgr_da_sess_new_result(IN da, IN da-session-handle, IN da-trans-handle, OUT session-status, OUT trans-status, OUT status)

   RT_C2Q_SESTAT: session status: either SMTP status code or an error code, e.g., connection refused etc.
7. qmgr_da_ta_new(IN da, IN da-session-handle, IN transaction, IN da-trans-handle, OUT trans-status, OUT status)

   - RT_Q2C_NTAID: new transaction id
   - RT_Q2C_MAIL: mail from
   - RT_Q2C_CDBID: cdb id

- RT_Q2C_RCPTIDX: rcpt index
- RT_Q2C_RCPT: rcpt to

The recipient data might be repeated to list multiple recipients Notice: we may run into a size limit of RCBs here; do we need something like a continuation RCB?.

8. qmgr_da_ta_new_result(IN da, IN da-session-handle, IN da-trans-handle, OUT trans-status, OUT status)

   See Section 3.9.2.1 about the status information that is sent from an SMTP client to QMGR.

   RT_C2Q_TASTAT: transaction status: this is the overall status of a transaction. It has two parts: an error code and an error state, the latter describes in which state of the SMTP dialogue (or internal state) the error occurred. An optional error message can be sent (RT_C2Q_STATT). If recipient statuses are sent then the record type RT_C2Q_TARSTAT is used instead. A recipient status consists of three items:

   - RT_C2Q_RCPT_IDX: rcpt index: this may be sent several times together
   - RT_C2Q_RCPT_ST: rcpt status (numeric error code)
   - RT_C2Q_RCPT_STT: text describing rcpt status.

9. qmgr_da_session_close(IN da, IN da-session-handle, OUT status)

   RT_Q2C_CSEID: close session id

10. qmgr_da_session_close_result(IN da, IN da-session-handle, OUT status)

    This value can be pretty much ignored for all practical purposes, except if we want to see whether the server behaves properly and still responds.

    RT_C2Q_SESTAT: session status (or do we want to use a different record type? Might be useful to distinguish to avoid confusion)

4.6.2 SMTP Client Implementation

4.6.2.1 SMTP Client Data Structures

The main SMTPC context structure looks like this:

```
struct sc_ctx_S
{
   unsigned int     sc_max_thrds;  /* Max number of threads */
   unsigned int     sc_wait_thrds; /* # of threads waiting to accept */
   unsigned int     sc_busy_thrds; /* # of threads processing request */
   unsigned int     sc_rqst_count; /* Total # of processed requests */
   uint32_t         sc_status;     /* SMTPC status */
   sm_str_P         sc_hostname;   /* SMTPC hostname */
   sc_t_ctx_P      *sc_scts;       /* array of sct's */
};
```

The last element of that structure is an array of SMTPC thread contexts (0 to `sc_max_thrds`−1):

```
struct sc_t_ctx_S
{
   sc_ctx_P         sct_sc_ctx;    /* pointer back to sc_ctx */
   unsigned int     sct_thr_id;    /* thread id (debugging) */
   unsigned int     sct_status;
   st_cond_t        sct_cond_rd;   /* received data from QMGR */
   sc_sess_P        sct_sess;      /* current session */
};
```

The condition variable denotes when data from the QMGR is received for this particular thread. The last element is a pointer to the SMTPC session:

```
struct sc_sess_S
{
   sc_t_ctx_P        scse_sct_ctx;  /* pointer to thread context */
   sm_file_T        *scse_fp;       /* file to use (SMTP) */
   sm_str_P          scse_rd;       /* smtp read buffer */
   sm_str_P          scse_wr;       /* smtp write buffer */
   sm_str_P          scse_str;      /* str for general use */
   sm_rpool_P        scse_rpool;
   unsigned int      scse_cap;      /* server capabilities */
   unsigned int      scse_flags;
   unsigned int      scse_state;
   struct in_addr   *scse_client;   /* XXX use a generic struct! */
   sc_ta_P           scse_ta;       /* current transaction */
   sessta_id_T       scse_id;
   sm_rcb_P          scse_rcb;      /* rcb for communication with QMGR */
   SOCK_IN_T         scse_rmt_addr; /* Remote address */
   st_netfd_t        scse_rmt_fd;   /* fd */
};
```

The SMTPC transaction structure looks as follows:

```
struct sc_ta_S
{
   sc_sess_P        scta_sess;       /* pointer to session */
   sm_rpool_P       scta_rpool;
   sc_mail_P        scta_mail;       /* mail from */
   sc_rcpts_P       scta_rcpts;      /* rcpts */
   sc_rcpt_P        scta_rcpt_p;     /* current rcpt for reply */
   unsigned int     scta_rcpts_rcvd; /* # of recipients replies received */
   unsigned int     scta_rcpts_tot;  /* number of recipients total */
   unsigned int     scta_rcpts_snt;  /* number of recipients sent */
   unsigned int     scta_rcpts_ok;   /* number of recipients ok */
   unsigned int     scta_rcpts_lmtp; /* #LMTP rcpts still to collect */
   unsigned int     scta_state;
   smtp_status_T    scta_status;     /* SMTP status code (if applicable) */
   sessta_id_T      scta_id;         /* transaction id */
   sm_str_P         scta_cdb_id;     /* CDB id */
};
```

4.6.2.2 SMTP Client Functions

4.6.2.2.1 Main Control In the `main()` function SMTPC calls several initialization function, one of which (`sc_init(sc_ctx)`) initializes the SMTPC context and allocates the array of SMTPC thread contexts. Then it starts the minimum number of threads (using `start_threads(sc_ctx)`) and the main thread takes care of signals afterwards. The threads run the function `sc_hdl_requests()` which receives the SMTPC context as parameter. This function looks for a free entry in the SMTPC thread context array, and allocates a new thread context which it assigns to that entry. It also allocates a new SMTPC session context. Thereafter it sets its status to `SC_T_FREE` and the first thread that is called informs the QMGR communication thread that SMTPC is ready to process tasks. The main part of the function processes a loop:

```
while (WAIT_THREADS(sc_ctx) <= max_wait_threads) { ... }
```

i.e., the thread stays active as long as the number of waiting threads is below the allowed maximum. This takes care of too many waiting threads by simply terminating them if the condition is false, in which case the thread cleans up after itself and terminates. Inside the loop the thread waits on its condition variable: `sc_t_ctx->sct_cond_rd`. If that wait times out, the current session (if one is open) will be terminated. If the QMGR actually has a task for this thread, then it first checks whether another thread should be started:

```
if (WAIT_THREADS(sc_ctx) < min_wait_threads &&
    TOTAL_THREADS(sc_ctx) < MAX_THREADS(sc_ctx))
{ /* start another thread */ }
```

and then handles the current session: `handle_session(sc_t_ctx)`. This functions handles one SMTP client session. The state of the session is recorded in `sc_sess->scse_state` and can take one of the following values:

SCSE_ST_NONE	no session active
SCSE_ST_NEW	new session
SCSE_ST_CONNECTED	connection succeeded
SCSE_ST_GREETED	received greeting
SCSE_ST_OPEN	connection open
SCSE_ST_CLOSED	close session

Based on this state the function opens a session if that hadn't happened yet and performs one transaction according to the data from the QMGR. Depending on a flag in `sc_sess->scse_flags` the session is optionally closed afterwards.

4.6.2.2.2 Communication Function As usual there is one thread that takes care of the communication between the module and the QMGR. This thread uses the following structure as context:

```
struct c2q_ctx_S
{
    sc_ctx_P        c2q_sc_ctx;    /* pointer back to SMTPC context */
    unsigned int    c2q_status;    /* status */
    st_netfd_t      c2q_fd;        /* fd for communication */
    unsigned int    c2q_sc_id;     /* smtpc id */
    st_cond_t       c2q_cond_rd;   /* cond.var for read */
```

```
    st_cond_t        c2q_cond_wr;   /* cond.var for write */
    unsigned int     c2q_maxses;    /* max. # of open sessions */
    sc_sess_P       *c2q_sess;      /* array of session ctx */
};
```

For initialization and termination the following two functions are provided:

```
sm_ret_T  sm_c2q_init(sc_ctx_P sc_ctx, c2q_ctx_P c2q_ctx, unsigned int sc_idx, unsigned int maxses);
sm_ret_T  sm_c2q_stop(c2q_ctx_P c2q_ctx);
```

The initialization function starts a thread that executes the function `void *sc_rcb_from_qmgr(void *arg)` which receives the c2q context as parameter. This function receives an RCB from QMGR and notifies a thread that is associated with the task or finds a free SMTPC thread if it is a new task. To maintain the former information one array for session contexts `c2q_sess` is allocated; its size is `maxses` which is set to `MAX_THREADS(sc_ctx)` by the caller. This allows the communication module to find the correct session context based on the session (or transaction) identifier sent by the QMGR in its requests if the request refers to an open session. To find a free SMTPC thread, the array `sc_scts` in the SMTPC context is searched for a `NULL` entry.

Status information can be sent back to the QMGR using the function `sm_ret_T sc_c2q(sc_t_ctx_P sc_t_ctx, uint32_t whichstatus, sm_ret_T ret, c2q_ctx_P c2q_ctx)`.

4.6.2.2.3 SMTP Related Functions The SMTP client functionality is fairly restricted right now, but the system implements full pipelining (in contrast to sendmail 8 which uses `MAIL` as synchronization point). As usual, the SMTP client is also able to speak LMTP.

To open and close a SMTP session two functions are provided: `sm_ret_T sc_sess_open(sc_t_ctx_P sc_t_ctx)` and `sm_ret_T sc_sess_close(sc_t_ctx_P sc_t_ctx)`. The function `sm_ret_T sc_one_ta(sc_t_ctx_P sc_t_ctx)` performs one SMTP transaction. As it can be seen from the prototypes, the only parameter passed to these function is the SMTPC thread context which contains (directly or indirectly) pointers to the current SMTPC session and transaction.

As shown in Section 4.6.2.1, the SMTPC session context contains three strings (see 3.16.7) that are used for the SMTP dialog and related operations.

Since the content database stores the mail in SMTP format, it can be sent out directly without any interaction. Similar to the SMTP server side this function access the file buffer directly to avoid too much copying.

4.7 Queue Manager

Just some items to take into consideration for the implementation of the queue manager. These are written down here so they don't get lost...

- multi-threaded
- must not block (pthreads ok?)

  Problem here: what about disk I/O? For example: calling fsync() for the logfile may cause the queue manager to block. If the thread implementation doesn't schedule another thread while one is blocked on disk I/O, then the entire process will hang and the queue manager will not respond to other requests.

If this actually happens (fairly likely on some OSs with user-land pthread implementation), and it causes a problem (performance), then it might be necessary to create another process that actually performs disk I/O on behalf of the QMGR.

- must scale (multiple CPUs)
- can we write some generic libraries?
- event driven (requests from SMTPS, feedback from SMTPC)
- event scheduling? which?
  - low activity: next queue run
  - group commits: answer within N ms

How about a flow diagram? Some architectural overview would be nice.

The QMGR should not have to deal with many connections. SMTPS and SMTPC are multi-threaded themselves; we may have some SMTPS/SMTPC processes. However, it won't be so many that we have a problem with the number of connections to monitor, i.e., poll() should be sufficient.

Which threading model should we choose? Just a few worker threads that will go back to work whenever they encounter a blocking action? See Section 3.20 for discussion.

Do we need priority queues or can we serve all jobs FIFO?

4.7.1 Queue Manager Implementation

The QMGR is based on the event threads library described in Section 4.3.8.

Currently access to tasks is controlled via the mutexes that control the queues: if a task is taken out of a queue, it is under the sole control of the thread that did it, no other thread can (should) access the task. Unless we change this access model, no mutex is necessary for individual tasks.

4.7.2 Locking

The queue manager has several data structures that can be concurrently accessed from different threads. Hence the access must be protected by mutexes unless there are other means which prevent conflicts. Some data structures can be rather larger, e.g., the various DBs and caches. Locking them for an extended time may cause lock contention. Some data structures and operations on them may allow to lock only a single element, others may require to lock the entire structure. Examples of the latter are adding and removing elements which in most cases require locking of the entire structure.

In some cases there might be ways around locking contention. For example, to delete items from a DB (or cache) the item might be just marked "Delete" instead of actually deleting it. This only requires locking of a single entry, not the entire DB. Those "Delete" entries can be removed in a single sweep later on (or during normal "walk through" operations), or they can be simply reclaimed for use. Question: what is more efficient? That is, if the DB is large and a walk through all elements is required to free a few then that might take too long, and we shouldn't hold a lock too long. We could gather "Delete" elements in a queue, then we don't have to walk through the entire DB. However, then the position of the elements must be fixed such that we can directly access and delete them, or at least lookup prior to deletion must be fast. If the DB internally may rearrange the location of entries then we can't keep a pointer to them. Question: will this ever happen? Some DB versions may do this, how about the ones we use? In some cases, some of the algorithm may require that DB elements don't move, but in most cases the elements just contain pointers to the data which isn't moved and hence can be accessed even if the DB rearranges its internal data structures.

4.7.2.1 Deadlock Avoidance

If a system locks various items then there is a potential for deadlocks. One way to prevent this is a locking hierarchy, i.e., items are always locked in the same order. We probably need to define a locking order. It's currently unclear how this can be done such that access is still efficient without too much locking contention. See also Section 4.7.2 for possible ways around locking contention.

4.7.3 Data Structures

The main context for QMGR looks like this: (2004-04-14)

```
struct qmgr_ctx_S
{
    sm_magic_T       sm_magic;
    pthread_mutex_t  qmgr_mutex;
    unsigned int     qmgr_status;   /* see below, QMGR_ST_* */
    time_T           qmgr_st_time;
    /* Resource flags */
    uint32_t         qmgr_rflags;   /* see QMGR_RFL_* */
    /* Overall value to indicate resource usage 0:free 100:overloaded */
    unsigned int     qmgr_total_usage;
    /* Status flags */
    uint32_t         qmgr_sflags;   /* see QMGR_SFL_* */
    sm_str_P         qmgr_hostname;
    sm_str_P         qmgr_pm_addr;  /* <postmaster@hostname> */
    /* info about connections? */
    fs_ctx_P         qmgr_fs_ctx;
    cdb_fsctx_P      qmgr_cdb_fsctx;
    unsigned long    qmgr_cdb_kbfree;
    edb_fsctx_P      qmgr_edb_fsctx;
    unsigned long    qmgr_edb_kbfree;
    unsigned long    qmgr_ibdb_kbfree;
    /* SMTPS */
    id_count_T       qmgr_idc;      /* last used SMTP id counter */
    int              qmgr_sslfd;    /* listen fd */
    int              qmgr_ssnfd;    /* number of used fds */
    uint32_t         qmgr_ssused;   /* bitmask for used elements */
    qss_ctx_P        qmgr_ssctx[MAX_SMTPS_FD];
    ssocc_ctx_P      qmgr_ssocc_ctx;
    occ_ctx_P        qmgr_occ_ctx;
    /* SMTPC */
    int              qmgr_sclfd;    /* listen fd */
```

```
    int              qmgr_scnfd;    /* number of used fds */
    uint8_t          qmgr_scused;   /* bitmask for used elements */
    qsc_ctx_P        qmgr_scctx[MAX_SMTPC_FD];
    sm_evthr_ctx_P   qmgr_ev_ctx;   /* event thread context */
    iqdb_P           qmgr_iqdb;     /* rsc for incoming edb */
    ibdb_ctx_P       qmgr_ibdb;     /* backup for incoming edb */
    sm_evthr_task_P  qmgr_icommit;  /* task for ibdbc commits */
    qss_opta_P       qmgr_optas;    /* open transactions (commit) */
    sm_evthr_task_P  qmgr_sched;    /* scheduling task */
    aq_ctx_P         qmgr_aq;       /* active envelope db */
    edb_ctx_P        qmgr_edb;      /* deferred envelope db */
    edbc_ctx_P       qmgr_edbc;     /* cache for envelope db */
    sm_evthr_task_P  qmgr_tsk_cleanup; /* task for cleanup */
    qcleanup_ctx_P   qmgr_cleanup_ctx;
    sm_maps_P        qmgr_maps;     /* map system context */
    /* AR */
    sm_evthr_task_P  qmgr_ar_tsk;   /* address resolver task */
    int              qmgr_ar_fd;    /* communication fd */
    qar_ctx_P        qmgr_ar_ctx;
    sm_rcbh_T        qmgr_rcbh;     /* head for RCB list */
    unsigned int     qmgr_rcbn;     /* number of entries in RCB list */
    /* currently protected by qmgr_mutex */
    qmgr_conf_T      qmgr_conf;
    sm_log_ctx_P     qmgr_lctx;
    sm_logconfig_P   qmgr_lcfg;
    uint8_t          qmgr_usage[QMGR_RFL_LAST_I + 1];
    uint8_t          qmgr_lower[QMGR_RFL_LAST_I + 1];
    uint8_t          qmgr_upper[QMGR_RFL_LAST_I + 1];
};
```

There are task contexts for QMGR/SMTPS (2004-04-15):

```
struct qss_ctx_S
{
    sm_magic_T       sm_magic;
    rcbcom_ctx_T     qss_com;
    qmgr_ctx_P       qss_qmgr_ctx;  /* pointer back to main ctx */
    int              qss_id;        /* SMTPS id */
    uint8_t          qss_bit;       /* bit for qmgr_ssctx */
    qss_status_T     qss_status;    /* status of SMTPS */
    unsigned int     qss_max_thrs;  /* upper limit for threads */
    unsigned int     qss_max_cur_thrs;      /* current limit for threads */
    unsigned int     qss_cur_session;       /* current # of sessions */
};
```

and QMGR/SMTPC (2004-04-15):

```
struct qsc_ctx_S
{
    sm_magic_T       sm_magic;
```

```
    rcbcom_ctx_T     qsc_com;
    qmgr_ctx_P       qsc_qmgr_ctx;  /* pointer back to main ctx */
    int              qsc_id;        /* SMTPC id */
    uint8_t          qsc_bit;       /* bit for qmgr_ssctx */
    dadb_ctx_P       qsc_dadb_ctx;  /* pointer to DA DB context */
    /* split this in status and flags? */
    qsc_status_T     qsc_status;    /* status of SMTPC */
    uint32_t         qsc_id_cnt;
};
```

Both refer to a generic communication structure:

```
struct qcom_ctx_S
{
    qmgr_ctx_P                qcom_qmgr_ctx; /* pointer back to main ctx */
    sm_evthr_task_P           qcom_tsk;      /* pointer to evthr task */
    sm_rcb_P                  qcom_rdrcb;    /* rcb for rd */
    SIMPLEQ_HEAD(, sm_rcbl_S) qcom_wrrcbl;   /* rcb list for wr */
    pthread_mutex_t           qcom_wrmutex;  /* protect qss_wrrcb */
};
```

The QMGR holds also the necessary data for SMTPS sessions (2004-04-15):

```
struct qss_sess_S
{
    sessta_id_T      qsses_id;
    time_T           qsses_st_time;
    sm_rpool_P       qsess_rpool;
    struct in_addr   qsess_client;  /* XXX use a generic struct! */
};
```

and transactions (2004-04-15):

```
struct qss_ta_S
{
    sm_rpool_P       qssta_rpool;
    time_T           qssta_st_time;
    qss_mail_P       qssta_mail;      /* mail from */
    qss_rcpts_T      qssta_rcpts;     /* rcpts */
    unsigned int     qssta_rcpts_tot; /* total number of recipients */
    unsigned int     qssta_flags;
    sessta_id_T      qssta_id;
    cdb_id_P         qssta_cdb_id;
    size_t           qssta_msg_size; /* KB */
    qss_ctx_P        qssta_ssctx;    /* pointer back to SMTPS ctx */
    pthread_mutex_t  qssta_mutex;
};
```

The open transaction context (from SMTPS) stores information about outstanding transactions, i.e., those transactions in SMTPS that have ended the data transmission, but have not yet been confirmed by the QMGR. This data structure (fixed size queue) is used for group commits to notify the threads in the SMTPS servers that hold the open transactions.

```
struct qss_opta_S
{
    unsigned int     qot_max;   /* allocated size */
    unsigned int     qot_cur;   /* currently used (basically last-first) */
    unsigned int     qot_first; /* first index to read */
    unsigned int     qot_last;  /* last index to read (first to write) */
    pthread_mutex_t  qot_mutex;
    qss_ta_P        *qot_tas;   /* array of open transactions */
};
```

Other structures that the QMGR currently uses are

- active queue (AQ, ACTEDB): `actdb_ctx_P qmgr_actdb;`
- SMTP server open connection cache: `ssocc_ctx_P qmgr_ssocc_ctx;`
- outgoing connection cache: `occ_ctx_P qmgr_occ_ctx;`
- incoming queue (IQDB, in memory): `iqdb_P qmgr_iqdb;`
- backup of incoming queue (IBDB, on disk): `ibdb_P qmgr_ibdb;`
- main queue: `edb_ctx_P qmgr_edb;` and a cache `edbc_ctx_P qmgr_edbc;`

All envelope DBs (INCEDB: ibdb and iqdb, ACTEDB, DEFEDB, EDB) have their own mutexes in their context structures.

IQDB contains references to `qss_sess_T`, `qss_ta_T`, and `qss_rcpts_T`.

The recipient structure in AQ uses these flags:

AQR_FL_IQDB	from IQDB
AQR_FL_DEFEDB	from DEFEDB
AQR_FL_SENT2AR	Sent to AR
AQR_FL_RCVD4AR	Received from AR
AQR_FL_RDY4DLVRY	Ready for delivery
AQR_FL_SCHED	Scheduled for delivery, is going to be sent to DA
AQR_FL_WAIT4UPD	Waiting for status update, must not be touched by scheduler
AQR_FL_TO	Too long in AQ
AQR_FL_TEMP	temporary failure
AQR_FL_PERM	permanent failure
AQR_FL_ARF	failure from SMAR
AQR_FL_DAF	failure from DA
AQR_FL_MEMAR	memory allocation for aqr_addrs failed, use fallback
AQR_FL_ARINCOMPL	addr resolution incomplete
AQR_FL_ARF_ADD	rcpt with SMAR failure added to delivery list
AQR_FL_TO_ADD	rcpt with timeout added to delivery list
AQR_FL_IS_BNC	this is a bounce
AQR_FL_IS_DBNC	double bounce
AQR_FL_DSN_PERM	perm error
AQR_FL_DSN_TMT	timeout
AQR_FL_DSN_GEN	bounce has been generated
AQR_FL_CNT_UPD	rcpt counters have been updated, i.e., aq_upd_ta_rcpt_cnts() has been called
AQR_FL_STAT_UPD	rcpt status (aqr_status) has been updated individually

4.7.4 Data Flow

Section 2.4.3.3 explains how transaction and recipient data flows through the various DBs in QMGR. This section tries to tie the various steps to functions in QMGR (which are explained in Section 4.7.5)

1. An envelope sender address (MAIL) is received by `sm_qss_react()` and stored in IQDB by `sm_q_ntaid()` which calls `iqdb_trans_new()` for that purpose.

2. An envelope recipient (RCPT) is received by `sm_qss_react()` and is stored in the incoming queue (IQDB) and in IBDB by `sm_q_rcptid()` which calls `iqdb_rcpt_add()` and `ibdb_rcpt_new()` to perform those actions.

3. When the final dot is received `sm_qss_react()` stores the content database information in the incoming queue (IQDB) and calls `sm_q_ctaid()` to store the transaction data (envelope sender and CDB id) in IBDB using `ibdb_ta_new()`. The data is copied into AQ by `aq_env_add_iqdb()` unless AQ is full in which case the entry is discarded from IBDB and a temporary error is returned.

4. Before mail reception is acknowledged the entire transaction data is safely committed to the backup of the incoming queue on disk (IBDB). This is achieved by the function `q_ibdb_commit()` which is invoked by the task `qmgr_ibdb_commit()` which can be triggered by `sm_q_schedctaid()`.

5. Recipient addresses that have been added to AQ from IQDB are sent to the address resolver by `qmgr_rcpt2ar()`, which is invoked indirectly from `aq_env_add_iqdb()` (see 3).

6. Results from the address resolver are received by `sm_qar_react()` which updates recipients in AQ. Possible results include (temporary) errors in which case the appropriate actions as explained in Section 4.7.5.7 are taken.

7. Delivery transactions consisting of one or more recipients are created by the scheduler `qmgr_sched()` based on various criteria and sent to delivery agents.

8. Delivery status received from delivery agents is used to update the data in the various queues (see Section 4.7.5.7 for details).

9. Recipient addresses which are waiting too long for a result from AR or DA must be removed from AQ, they are put into DEFEDB with a temporary error, unless the overall queue return timeout is exceeded.

10. Data from DEFEDB is used to feed the active queue; entries are read from it based on their "next time to try" (or whatever criteria the scheduler wants to apply).

4.7.5 Functions

The `main()` function of the QMGR is very simple (Notice: in almost all example code error checking etc has been removed for simplicity).

```
ret = sm_qmgr_init0(qmgr_ctx);  /* basic initialization */
ret = sm_qmgr_rdcf(qmgr_ctx);   /* read configuration */
ret = sm_qmgr_init(qmgr_ctx);   /* initialization after configuration */
ret = sm_qmgr_start(qmgr_ctx);  /* start all components */
ret = sm_qmgr_loop(qmgr_ctx);   /* start event threads loop */
ret = sm_qmgr_stop(qmgr_ctx);   /* stop all componets */
```

where all functions do what is obvious from their name.

The main loop `sm_qmgr_loop()` simply calls `evthr_loop(qmgr_ctx->qmgr_ev_ctx)`.

4.7.5.1 Initialization Functions

`sm_qmgr_init0()` performs basic initialization, `sm_qmgr_rdcf()` reads the configuration (currently (2004-02-13) only command line parameters), and `sm_qmgr_init()` initializes various QMGR data structures.

`sm_qmgr_start()` starts various tasks:

```
ret = sm_qm_stli(qmgr_ctx);
ret = sm_qm_stcommit(qmgr_ctx, now);
ret = sm_qm_stsched(qmgr_ctx, now);
```

`sm_qm_stli()` starts two (event thread) tasks listening for connections from SMTPS and SMTPC using the function `sm_qm_smtpsli()` and `sm_qm_smtpcli()`. `sm_qm_stcommit()` starts the periodic commit task and `sm_qm_stsched()` starts the scheduling task.

4.7.5.2 Communication Functions

The two listener tasks `sm_qm_smtpsli()` and `sm_qm_smtpcli()` do basically the same: wait for a new connection from the respective service (SMTPS/SMTPC), "register" it in the QMGR context, and start one task `sm_qmgr_smtp`X`(sm_evthr_task_P tsk)` that takes care of the communication with the SMTPX process. Notes:

- the listener tasks start one task per SMTPS/SMTPC process. Hence it is important that there are not too many processes (currently there is a fixed size list in the QMGR context), because otherwise there might be too much I/O activity to check (if it's one process per connection then there would be a lot of open filedescriptors used by the system, with all the drawbacks mentioned earlier on).
- `sm_qm_smtpcli()` may go away since the QMGR is supposed to request delivery agents.

The communication tasks `sm_qmgr_smtp`*X*`()` dispatch a read function `sm_smtp`*X*`2qmgr()` or a write function `sm_qmgr2smtp`*X* to deal with the communication request. Those functions use the read RCB `qsX_rdrcb` to read (sequentially) data from SMTPS/SMTPC and a list of write RCBs `qsX_wrrcbl` to write data back to those modules. Access to the latter is protected by a mutex and RCBs are appended to the list by various functions. The communication tasks are activated via read/write availability, where the write availability is additionally triggered by functions that put something into the list of write RCBs (otherwise the task would be activated most of the time without actually having anything to do).

The read functions `sm_smtp`*X*`2qmgr()` receive an RCB `qs`*X*`_ctx->qs`*X*`_rdrcb` from the module and then call the function `sm_qs`*X*`_react()` to decode the RCB and act accordingly. Those functions may return different value to determine what should happen next with the task. If it is an error, the task terminates (which might be overkill), other values are: `QMGR_R_WAITQ` (translated to `EVTHR_WAITQ`), `QMGR_R_ASYNC` (translated to `EVTHR_OK`), `EVTHR_DEL` which cause the task to terminate; other values are directly returned to the event threads library. `QMGR_R_ASYNC` means that the task has already been returned to the event thread system (waitq), see Section 3.20.5.1.3.

The write function `sm_qmgr2mod()` locks the mutex `qs`*X*`_wrmutex`, then checks whether the list `qs`*X*`_wrrcbl` of RCBs is empty. If it is, then the task returns and turns off the WRITE request. Otherwise it sends the first element to the respective module using `sm_rcb_snd()`, removes that element and if the list is empty thereafter turns off the WRITE when it returns. Notice: it currently does not go through the entire list trying to write it all. This is done to prevent the thread from blocking, it is assumed that a single RCB can be sent. This might be wrong in which case the thread blocks (and hopefully another runs), which might be prevented by requiring enough space in the communication buffer (can be set via `setsockopt()` for sockets).

4.7.5.3 Commit Task

The commit task `sm_qm_stcommit()` is responsible for group commits. It checks the list of open transactions `qmgr_ctx->qmgr_optas` and if it isn't empty calls `q_ibdb_commit(qmgr_ctx)` which in turns commits the current INCEDB and then notifies all outstanding transactions of this fact. This is done by going through the list and adding an RCB with the commit information to the list of RCBs `qss_wrrcbl` for the task `qss_ta->qssta_ssctx` that handles the transaction `qss_ta`.

4.7.5.4 Scheduler

The scheduling function `sm_qm_stsched()` is supposed to implement the core of the QMGR.

A recipient goes through the following stages:

1. it is created in AQ (either from IQDB or DEFEDB; a flag indicates the source);
2. the address is sent to SMAR (unless already available from DEFEDB);
3. the resolved address is added to the recipient structure (unless already available from DEFEDB);

4. the item is scheduled for delivery, i.e., an RCB is created;
5. when the RCB is sent to a DA another flag is set;
6. the status is updated when the results are received from the DA;

4.7.5.5 QMGR to SMTPC Protocol

The function `sm_qs2c_task(qsc_ctx_P qsc_ctx, aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, sm_rcbe_P rcbe, sessta_id_P da_sess_id, sessta_id_P da_ta_id)` creates one session with one transaction for SMTPC. The protocol is as follows:

RT_Q2C_ID	SMTPC identifier
RT_Q2C_DCID	delivery class identifier
RT_Q2C_ONESEID	Session id, only one transaction (hack)
RT_Q2C_SRVIP4	IPv4 address of server (hack)
RT_Q2C_NTAID	New transaction id
RT_Q2C_MAIL	Mail from address
RT_Q2C_CDBID	CDB identifier
RT_Q2C_RCPT_IDX	recipient index
RT_Q2C_RCPT	recipient address

Additional recipients can be added via `sm_qs2c_add_rcpt(qsc_ctx_P qsc_ctx, aq_rcpt_P aq_rcpt, sm_rcbe_P rcbe)` which justs adds recipient index and address to the RCB.

If the transaction denotes a bounce message only one recipient can be send and instead of the record tag `RT_Q2C_NTAID` either `RT_Q2C_NTAIDB` (bounce) or `RT_Q2C_NTAIDDB` (double bounce) is used. Additionally an entire error text is sent using `RT_Q2C_B_MSG` (bounce message) as record tag. Currently this does not include the headers. It should be something like:

```
Hi! This is the sendmail X MTA. I'm sorry to inform you that a mail
from you could not be delivered. See below for details.
```

listing recipient address, delivery host, and delivery message for each failed recipient.

4.7.5.6 Load Control Implementation

4.7.5.6.1 Load Control Implementation: Data Structures The main QMGR context contains three arrays which store the lower and upper thresholds for various resources and the current usage. A single scalar contains the overall resource usage.

```
uint8_t  qmgr_usage[QMGR_RFL_LAST_I + 1];
uint8_t  qmgr_lower[QMGR_RFL_LAST_I + 1];
uint8_t  qmgr_upper[QMGR_RFL_LAST_I + 1];

/* Overall value to indicate resource usage 0:free 100:overloaded */
unsigned int  qmgr_total_usage;
```

To store the amount of free disk space, two data structures are used: one to store the amount of available disk space per partition (see also Section 3.4.10.13.1):

```
struct filesys_S {
        dev_t           fs_dev;       /* unique device id */
        unsigned long   fs_kbfree;    /* KB free */
        unsigned long   fs_blksize;   /* block size, in bytes */
        time_T          fs_lastupdate; /* last time fs_kbfree was updated */
        const char     *fs_path;      /* some path in the FS */
};
```

and one which contains an array of those individual structures:

```
struct fs_ctx_S {
#if SM_USE_PTHREADS
    pthread_mutex_t  fsc_mutex;
#endif /* SM_USE_PTHREADS */
    int              fsc_cur_entries; /* cur. number of entries in fsc_sys*/
    int              fsc_max_entries; /* max. number of entries in fsc_sys*/
    filesys_P        fsc_sys;       /* array of filesys_T */
};
```

4.7.5.6.2 Load Control Implementation: Functions The function `qm_comp_resource(qmgr_ctx_P qmgr_ctx, thr_lock_T locktype)` computes a value that is a measure for the overall resource usage: `qmgr_total_usage`. Moreover, the function also invokes functions that return the amount of free disk for a DB that is stored on disk: `cdb_fs_getfree()`, `edb_fs_getfree()`, and `ibdb_fs_getfree()`. Each of these functions receives a pointer to a variable of type `fs_ctx_T` and a pointer to a integer variable which will contain the amount of available disk space after a succesful return. The functions themselves check the last update timestamp to avoid invoking system functions too often. Since each DB operations tries to keep track of the amount of disk space changes, this should return a reasonable estimate of the actual value.

The function `q2s_throttle(qss_ctx_P qss_ctx, sm_evthr_task_P tsk, unsigned int nthreads)` informs one SMTP server (referenced by `qss_ctx`) about the new maximum number of threads it should allow.

The generic function `qs_control(qss_ctx_P qss_ctx, int direction, unsigned int use, unsigned int resource)` checks the new usage of a resource and based on the input parameter `direction` decides whether to (un)throttle one SMTP server. `qs_control()` has the following behavior: throttle the system iff

- asked for and
- the use of the resource exceeds the upper threshold and
- the current maximum number of threads is greater than 0

else unthrottle the system iff

- asked for and
- the use of the resource is below the lower threshold and
- the current maximum number of threads is less the allowed maximum and

- the system is not in state OK and
- there is no resource shortage at all

The specific function `qs_unthrottle(qss_ctx_P qss_ctx)` checks whether one SMTP server can be unthrottled based on the current resource usage. It is called by `sm_smtps_wakeup()` which is scheduled by `sm_qss_wakeup(qmgr_ctx_P qmgr_ctx, thr_lock_T locktype)` as a `sleep()` task. `sm_qss_wakeup()` in turn is invoked from `qm_resource()` when all resources are available (again).

The function `qs_comp_control(qss_ctx_P qss_ctx, bool unthrottle)` is invoked from `sm_qss_react()`. It will only check whether the address resolver (SMAR) is available and accordingly call `qs_control()`.

4.7.5.7 Updating Recipient Status

The requirements for updating the recipient status after a delivery attempt has been made are described in Section 2.4.3.4. Section 3.4.16 describes the the functionality, which distinguishes several reasons that require updating the status of a recipient:

1. after receiving the routing information from SMAR: this may cause a (temporary) error;
2. after receiving status information from a delivery agent. There are two subcases:
   (a) some recipients have a return code that differs from the return code for the complete transaction, i.e., a recipient had an error (success does not need to be returned, in that case the transaction return code applies),
   (b) there is one return code for the complete transaction.

   Note: there is always a return code for the complete transaction, even if all recipients failed, that code will a temporary error if at least one of the recipient had only a temporary error, otherwise it's a permanent failure (the "smallest" failure).
3. an entry is too long in the active queue.

Before examining these cases, a short note about updating the various queues: entries in IQDB are removed immediately if the recipient was in that queue (this can be done because the recipient is safely stored in DEFEDB or IBDB). To update DEFEDB and IBDB more complicated measures are taken: a request is queued that the status must be changed (this may also mean removal of an entry from the respective DB) while the function goes through all the recipients of the transaction. DEFEDB provides functions to do this: `edb_ta_rm_req()` and `edb_rcpt_rm_req()` which are described in Section 4.10.4. See Section 4.10.2 about the implementation of updating IBDB based on a list of change requests. However, if a recipient was successfully delivered on the first attempt, IBDB can be updated directly: even if further operations fail (which are related to updating the recipient status), the recipient can be removed without any negative consequences.

4.7.5.7.1 Preserving Order of Updates As explained in Section 3.4.16.1 it is necessary to preserve the order of updates for recipients and transactions when those changes are committed to DEFEDB.

4.7.5.7.2 qm-fr-sc-rcpts To update the status for some (failed) recipients (case 2a) the function

`qm_fr_sc_rcpts(qmgr_ctx_P qmgr_ctx, sessta_id_T da_ta_id, sm_rcb_P rcb, unsigned int err_st)`

is used, the RCB contains the recipient status from a DA. This function simply takes the data out of the RCB and updates the recipient status in the active queue. For this it invokes

`aq_rcpt_status(aq_ctx, da_ta_id, idx, rcpt_status, err_st, errmsg)`,

which updates the field `aqr_status_new` that is later on used for `aq_upd_ta_rcpt_cnts()` (see 4.7.5.7.6) which requires the previous and the new status of a recipient to determine which recipients counters to change in the transaction context.

4.7.5.7.3 qda-update-ta-stat To update the status for an entire transaction (case 2b) from a DA the function

`qda_update_ta_stat(qmgr_ctx_P qmgr_ctx, sessta_id_T da_ta_id, sm_ret_T status, unsigned int err_st, dadb_ctx_P dadb_ctx, dadb_entry_P dadb_entry, aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, thr_lock_T locktype)`

is called. This function walks through all recipients of a transaction and updates the various DBs and counters based on the individual recipient status (which may be different from the overall transaction status). See Section 2.4.3.4 for a high-level description.

The function `qda_update_ta_stat()` simple invokes

`qda_upd_ta_stat(qmgr_ctx, da_ta_id, status, err_st, dadb_ctx, dadb_entry, aq_ta, aq_rcpt, &edb_req_hd, &ibdb_req_hd, locktype)`

(see 4.7.5.7.4) and then writes the changes for DEFEBD and IBDB to disk (unless the result is an error).

4.7.5.7.4 qda-upd-ta-rcpt-stat The function

`qda_upd_ta_rcpt_stat(qmgr_ctx_P qmgr_ctx, sessta_id_T da_ta_id, sm_ret_T status, unsigned int err_st, dadb_ctx_P dadb_ctx, dadb_entry_P dadb_entry, aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, edb_req_hd_P edb_req_hd, ibdb_req_hd_P ibdb_req_hd, thr_lock_T locktype)`

can be used to update an entire transaction, i.e., all recipients of that transaction, or just an individual recipient. These two cases are distinguished by specifying exactly one of the DA transaction identifier `da_ta_id` (i.e., the id must be valid – the first character must not be '\0') and the recipient `aq_rcpt` (i.e., must not be NULL).

This function is also used in other places to update the status of a single recipient, e.g., for failures from the address resolver (called from the scheduler when it comes across such a recipient). `qda_upd_ta_rcpt_stat()` invokes

`q_upd_rcpt_stat(qmgr_ctx, ss_ta_id, status, err_st, aq_ta, aq_rcpt, edb_req_hd, ibdb_req_hd, &iqdb_rcpts_done, THR_NO_LOCK)`.

(see 4.7.5.7.5) for all recipients that need to be updated. Afterwards it checks whether `iqdb_rcpts_done` is greater than zero in which case the function `qda_upd_iqdb(qmgr_ctx, iqdb_rcpts_done, ss_ta_id, cdb_id, ibdb_req_hd)` is invoked, see 4.7.5.7.10.

If there are no deliverable recipients in AQ anymore for the current transaction or it is required to update the transaction, then the function performs the following steps: first check whether there are no recipients at all, i.e., `aq_ta->aqt_rcpts_left` is zero, which means that the transaction and the data file (CDB) must be removed. If that's not the case but the transaction needs to be updated in DEFEDB, then a request is appended to the DEFEDB request list and the flag `AQ_TA_FL_DEFEDB` is set[3] and the flags `AQ_TA_FL_EDB_UPD_C` and `AQ_TA_FL_EDB_UPD_R` are cleared. A transaction needs to be updated if at least one of the following conditions helds:

- When a RCPT is written to/removed from DEFEDB (R) and the counters in the TA change (C) or the TA is not in DEFEDB (D): $R \wedge (C \vee \neg D)$

  Note: When a RCPT is updated in DEFEDB then TA is in DEFEDB or the counters change (the counters do not change iff the RCPT status does not change; if the RCPT status does not change, then the only reason the RCPT is written to DEFEDB is because it was there earlier and hence TA was there too – that is a pre-requirement of the algorithm: a RCPT is only in DEFEDB iff its TA is there too). Hence we can simplify $R \wedge (C \vee \neg D)$ to $R \wedge C$.

  Note: this is a side-effect of the current scheduler which keeps recipients in AQ until a delivery attempt is complete. If the scheduler changes to include pre-empting then the update logic must be modified to take care of that, i.e., the requirement – a RCPT is only in DEFEDB iff its TA is there too – does not necessarily hold anymore.

- When the counters in TA change (C) and it is in DEFEDB (D): $C \wedge D$

This can be expressed as: $(R \wedge C) \vee (C \wedge D) \equiv C \wedge (R \vee D)$

Without the simplification it is: $(R \wedge (C \vee \neg D)) \vee (C \wedge D) \equiv (R \wedge C) \vee (R \wedge \neg D) \vee (C \wedge D) \equiv (C \wedge (R \vee D)) \vee (R \wedge \neg D)$

If `aq_ta->aqt_rcpts_left` is zero and the transaction is in DEFEDB, then a remove request is appended to the request list.

If there are no more recipients in AQ for the TA (`aq_ta->aqt_rcpts == 0`), then the TA is removed from AQ.

If `aq_ta->aqt_rcpts_left` is zero and the CDB identifier is set (which must be the case), then the entry is removed from the CDB.

Finally, if the DA TA identifier is valid and the DA context is not NULL, then the session is closed (which can be done because the scheduler is currently a hack that only uses one transaction per session).

4.7.5.7.5 q-upd-rcpt-stat

The function

```
q_upd_rcpt_stat(qmgr_ctx_P qmgr_ctx, sessta_id_T ss_ta_id, sm_ret_T status, unsigned int err_st,
aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, edb_req_hd_P edb_req_hd, ibdb_req_hd_P ibdb_req_hd, unsigned
int *piqdb_rcpts_done, thr_lock_T locktype)
```

in turn updates the status for one recipient. If the recipient is in IQDB and it won't be retried, i.e.,

```
(rcpt_status == SM_SUCCESS || smtp_reply_type(rcpt_status) == SMTP_RTYPE_FAIL || !AQR_MORE_DESTS(aq_rcpt)
|| AQR_DEFER(aq_rcpt))
```

then it is immediately removed from IQDB. Next the recipient counters in the transaction are updated:

```
aq_upd_ta_rcpt_cnts(aq_ta, aq_rcpt->aqr_status, rcpt_status)
```

[3]Note: this is too early, the flag should be set after the transaction has been committed to DEFEDB, but that's hard to accomplish because it would require to have another list of transactions which would be updated after DEFEDB has been successfully changed.

(see 4.7.5.7.6).

Then one of two functions is called:

1. `q_upd_rcpt_ok(qmgr_ctx, ss_ta_id, aq_ta, aq_rcpt, &ibdb_rcpt, rcpt_id, edb_req_hd)`

   if the recipient has been delivered or is a double bounce that can't be delivered (and hence will be dropped on the floor); see 4.7.5.7.7.

2. `q_upd_rcpt_fail(qmgr_ctx, ss_ta_id, rcpt_status, aq_ta, aq_rcpt, &ibdb_rcpt, rcpt_id, edb_req_hd, ibdb_req_hd)`

   for temporary or permanent errors; see 4.7.5.7.8.

Afterwards it is checked whether there can be no more retries for that recipient in which case it is removed from AQ[4], otherwise the next destination host will be tried and the flags `AQR_FL_SCHED`, `AQR_FL_WAIT4UPD`, `AQR_FL_STAT_NEW`, and `AQR_FL_ERRST_UPD` are cleared.

4.7.5.7.6 aq-upd-ta-rcpt-cnts

The counters in the transaction are updated via

`aq_upd_ta_rcpt_cnts(aq_ta, oldstatus, newstatus)`

This function sets the flag `AQ_TA_FL_EDB_UPD_C` if a counter has been changed.

4.7.5.7.7 q-upd-rcpt-ok

Case 1 (from 4.7.5.7.5): `q_upd_rcpt_ok()` is responsible to remove a recipient from the queues in which it is stored.

`q_upd_rcpt_ok(qmgr_ctx_P qmgr_ctx, sessta_id_T ss_ta_id, aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, ibdb_rcpt_P ibdb_rcpt, rcpt_id_T rcpt_id, edb_req_hd_P edb_req_hd)`

In case of a double bounce it decrements the number of recipients left and logs the problem (dropped a double bounce). Then it removes the recipient from IBDB if it is stored there (directly, without going via the request queue), or from DEFEDB by appending the remove operation to the request queue. Finally, if the recipient is a (double) bounce, the function `qda_upd_dsn()` is called to remove the recipients for which the DSN has been generated; see 4.7.5.7.9.

4.7.5.7.8 q-upd-rcpt-fail

Case 2 (from 4.7.5.7.5): `q_upd_rcpt_fail()` examines `rcpt_status` to decide whether it is a temporary error or a permanent failure. In the former case the time in the queue is checked: if it exceeds a limit and there are no more destination hosts to try or the recipient must be deferred (e.g., address resolver failure), then two flags are set: `AQR_FL_PERM` and `AQR_FL_DSN_TMT`, in the latter case the flags `AQR_FL_PERM` and `AQR_FL_DSN_PERM` are set.

If the recipient can't be delivered and is not a double bounce itself then `qm_bounce_add(qmgr_ctx, aq_ta, aq_rcpt, errmsg)` is called to create a bounce message for this recipient; see 4.7.5.8.1.

If there are no more destinations to try or the recipient must be deferred (because of an address resolver problem or because it is too long in AQ), or a bounce message has been generated[5], then the number of tries is incremented, the next time to try is computed if necessary (i.e., recipient has only a temporary failure or it has a permanent failure but no bounce because generation of bounce recipient failed), and

[4] this is ok because the recipient is in a persistent DB already, so even if the update requests for DEFEDB or IBDB fail the recipient can be recovered from either DB.

[5] the latter case is not really necessary but a result of the current bounce handling implementation, e.g., `qda_upd_dsn()` assumes that the recipients are in DEFEDB.

a request to update the recipient status in DEFEDB is appended to the request list. If that was successful and the recipient is a (double) bounce then `qda_upd_dsn(qmgr_ctx, aq_ta, aq_rcpt, ss_ta_id, edb_req_hd)` is called to remove the recipients for which this was a bounce (see 4.7.5.7.9).

If the recipient must be retried, i.e., it is not permanent failure then it is added to the EDB cache: `edbc_add(qmgr_ctx->qmgr_edbc, rcpt_id, aq_rcpt->aqr_next_try, false)`

If the recipient was in IQDB then a status update is appended to the request list for IBDB using the function `ibdb_rcpt_app()`.

Finally `q_upd_rcpt_fail()` returns a flag value that indicates either an error or whether some actions (in this case: activate the address resolver) need to be performed by the caller.

4.7.5.7.9 qda-upd-dsn

`qda_upd_dsn(qmgr_ctx_P qmgr_ctx, aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, sessta_id_T ss_ta_id, edb_req_hd_P edb_req_hd)`

is responsible for removing the recipients for which the DSN has been generated, which is done by going through its list (`aq_rcpt->aqr_dsns[]`) and appending remove requests to the DEFEDB change queue. It also updates the number of recipients left if necessary, i.e., if the DSN was for more than one recipient, and resets the used data structures.

4.7.5.7.10 qda-upd-iqdb

`qda_upd_iqdb(qmgr_ctx_P qmgr_ctx, unsigned int iqdb_rcpts_done, sessta_id_T ss_ta_id, cdb_id_P cdb_id, ibdb_req_hd_P ibdb_req_hd)` updates IQDB status for one transaction; if all recipients have been delivered then it removes the transaction from IQDB using `iqdb_trans_rm()`, adds a request to remove it from IBDB via `ibdb_ta_app()` and removes it from the internal DB using `qss_ta_free()`.

4.7.5.8 Handling Bounces

The current implementation of sendmail X does not support DSN per RFC 1894, but it creates non-delivery reports in a "free" format; see also 4.7.5.5.

4.7.5.8.1 qm-bounce-add

If a bounce message is generated the function

`qm_bounce_add(qmgr_ctx_P qmgr_ctx, aq_ta_P aq_ta, aq_rcpt_P aq_rcpt, sm_str_P errmsg)`

is used. See Section 4.10.3 about the data structures that are relevant here (AQ transaction and recipient), and Section 4.7.3 about the flags (esp. those containing DSN or BNC in the name).

To generate a bounce, a new recipient is created (the "bounce recipient") using the function `qm_bounce_new()` unless the transaction already has a bounce recipient (that hasn't been scheduled yet). This recipient has an array `aqr_dsns` which contains the indices of the recipients for which this recipient contains the bounce message. Whether a transaction already has a (double) bounce recipient is recorded in the transaction (see 4.10.3): `aqt_bounce_idx` and `aqt_dbl_bounce_idx`. These can be reused to add more recipients to a bounce recipient (instead of sending one DSN per bounce).

4.7.5.8.2 qm-bounce-new

The function

`qm_bounce_new(qmgr_ctx_P qmgr_ctx, aq_ta_P aq_ta, bool dbl_bounce, aq_rcpt_P *aq_rcpt_bounce)`

creates a new bounce recipient. It uses `aq_ta->aqt_nxt_idx` as the index for the bounce recipient (after checking it against the maximum value: currently the index is only 16 bits) and stores the value in `aqt_bounce_idx` or `aqt_dbl_bounce_idx`, respectively. `aq_rcpt_add()` is used to add a new recipient to AQ, then an array of size `aq_ta->aqt_rcpts_tot` is created to hold the indices of those recipients for which this will be a bounce. This array is in general too big, some optimization can be applied (later on). `qm_bounce_new()` then fills in the data for the recipient and sends it to the address resolver using `qmgr_rcpt2ar()`. It also increases the number of recipients for this transaction (`aqt_rcpts_tot` and `aqt_rcpts`). This may create an inconsistent state since the bounce recipient is only in AQ, not in a persistent DB (DEFEDB), see 4.7.5.8.3.

4.7.5.8.3 Bounce Recipients are only created in AQ A bounce recipient is not written to a persistent DB when it is generated, but the failed recipients are written to DEFEDB. Only when a delivery attempt for a bounce message fails the bounce recipient is written to DEFEDB and the recipients for which it is a bounce are removed by `qda_upd_dsn()`, see 4.7.5.7.9. Hence when the system crashes before the bounce is delivered (or at least tried and then written to the queue), the bounce will be lost. However, the original recipients that caused the bounce are in DEFEDB, and hence the bounce message can be reconstructed.

Alternatively the bounce recipient can be written to one of the persistent queues and the original recipients can be removed, this could reduce the on-disk storage. However, it requires that the RCB to store the bounce recipient is fairly large because it contains the complete error text for each failed recipient and a large list of recipient indices. Theoretically it might also be possible to store the error text in IBDB, but that most likely requires changes to the storage format which does not make much sense because bounces should occur infrequently. Moreover, recovery of IBDB would become more complicated. Additionally, the failed recipient might not be in IBDB but in DEFEDB, hence making it even harder to correctly reconstruct the bounce data because it can be spread out over various places.

This is a place where optimizations are certainly possible, but it is currently not important enough (it is more important to implement the full sendmail X.0 system instead of optimizing bounces).

4.7.5.9 DSN: Delayed

As explained in Section 2.4.6.3 there are 8 states with respect to delayed DSNs which means 3 bits are required for a recipient to denote those states.

1. did not request dDSN: nothing to do.
2. did request dDSN:
   (a) no dDSN generated (yet): in case of a temporary error the timeout needs to be checked. If it is exceeded, then generate a dDSN and proceed to state 2(b)i.
   (b) dDSN generated.
      i. dDSN not yet delivered: as soon as the dDSN has been delivered, all recipients for which this was a dDSN must be updated accordingly (state 2(b)ii). This means that entries in DEFEDB need to be changed after a successful delivery, which causes extra disk I/O. *Question*: can we avoid this extra I/O? We could keep the dDSN in AQ and update the information the next time the recipient is read into AQ. This costs a bit of memory but avoids disk I/O and is therefore preferred. When a recipient is read from DEFEDB and it is in state 2(b)i then check AQ about the dDSN.
         A. If it's there and has been delivered then change the state of the recipient to 2(b)ii. Moreover, remove this recipient from the list of recipients in dDSN and if that list becomes empty then remove the dDSN itself.

B. If it's there and has not yet been delivered then we have a problem: we need to wait to the end of the delivery attempt to see what kind of updates need to be done. This is non-trivial.

C. If the dDSN is not in AQ then change the state back to 2a.

There are more cases to consider: delivery of the dDSN may fail (temporarily or permanently). In that case the recipients must be updated too. Could we store some information in EDBC? Or just keep a list of recipient-ids around in which stored/delivered dDSNs are listed? That is not that simple because it requires cross-references (dDSN to recipient, recipient to dDSN) for bookkeeping (when to remove an entry from that list, i.e., when are all referenced recipients updated).

ii. dDSN delivered: done.

The complexity of updating recipients in case 2(b)i is very high. That can cause programming errors and maybe inconsistencies in AQ or DEFEDB. *Question*: Is there are simpler approach to the problem?

Here is one proposal: write the dDSN to DEFEDB and immediately change the state to 2(b)ii, as the dDSN will be safely stored. Note: if updating DEFEDB fails then the recipient for which this is a dDSN is not written to DEFEDB either, hence this approach should be safe. Moreover, it is significantly simpler, and causes only a bit extra disk I/O because the recipient is written to DEFEDB anyway. This approach causes the handling of bounces and dDSNs to be different, which however should not matter much as the approach explained above is definitely different from bounce handling. However, this approach causes another problem: dDSN will be sent individually; see Section 2.4.6.

Why are dDSNs and bounces ("failure" DSNs: fDSN) so different? A fDSN means that the original recipients will be removed while they will be retried for a dDSN. So for a fDSN DEFEDB must be changed anyways (recipients must be removed), but for a dDSN it could (theoretically) be avoided.

4.7.5.10 AR to QMGR

The address resolver sends resolved addresses to QMGR which are used in turn by the scheduler for delivery attempts. The basic protocol returns a DA that must be used for delivery and a number of addresses to try.

A significantly more complicated part is alias expansion (see also Section 3.4.15.1).

4.8 Address Resolver Implementation

The address resolver is called `smar` for *s*end*m*ail *a*ddress *r*esolver since `ar` is already used. SMAR is based on the event threads library described in Section 4.3.8.

The description below is based on the implementation from 2003-06-26 and hence not up to date.

The AR uses a very simple *mailertable* which must be in a strict form: a domain part of an e-mail address, then one or more whitespace characters, followed by the IPv4 address (in square brackets) of the host to which the mail should be sent or the hostname itself. If an entry is not found in the mailertable or the RHS is a hostname, DNS lookups (for MX and A records) are performed (see Section 4.3.10 for the DNS library).

The main context for the AR has currently the following elements:

```
struct smar_ctx_S
```

```
{
        sm_magic_T        sm_magic;
        pthread_mutex_t   smar_mutex;
        int               smar_status;   /* see below, SMAR_ST_* */
        sm_evthr_ctx_P    smar_ev_ctx;   /* event thread context */
        int               smar_qfd;      /* fd for communication with QMGR */
        sm_evthr_task_P   smar_qmgr_tsk; /* QMGR communication task */
        rcbcom_ctx_T      smar_qmgr_com; /* QMGR communication context */
        sm_log_ctx_P      smar_lctx;
        ipv4_T            smar_nameserveripv4;
        unsigned int      smar_dns_flags;
        dns_tsk_P         smar_dns_tsk;
        dns_mgr_ctx_P     smar_dns_mgr_ctx;
        bht_P             smar_mt;       /* "mailertable" (XXX HACK) */
};
```

4.8.1 Resolving Recipient Addresses

To store addresses sent by the QMGR to the AR for resolving, the following structure is used:

```
struct smar_rcpt_S
{
        sm_str_P          arr_rcpt_pa;   /* printable addr */
        rcpt_id_T         arr_rcpt_id;   /* rcpt id */
        sm_str_P          arr_domain_pa;
        unsigned int      arr_flags;     /* status of address resolving */
        unsigned int      arr_da;        /* DA */
        int               arr_n_mx;      /* total number of MX records */
        int               arr_c_mx;      /* current number of MX records */
        int               arr_n_a;       /* total number of A records */
        smar_dns_T       *arr_res;       /* array of results */
        ipv4_T            arr_ip4;       /* single A record */
        sm_rcbe_P         arr_rcbe;      /* RCB to write back result */
        smar_ctx_P        arr_smar_ctx;  /* pointer back to SMAR context */
};
```

`arr_n_mx` stores the total number of MX records after the initial query for an MX record comes back. `arr_c_mx` keeps track of the current number of answer for those MX records, i.e., when both variables have the same value then all outstanding requests have been answered and the complete result can be returned to the QMGR.

The array of results has the following type:

```
struct smar_dns_S
{
        unsigned int      ardns_ttl;     /* TTL from DNS */
        unsigned short    ardns_pref;    /* preference from DNS */
        sm_cstr_P         ardns_name;    /* name from DNS */
        int               ardns_n_a;     /* number of A records */
        ipv4_T           *ardns_a;       /* pointer to list of A records */
```

```
};
```

The address resolver receives recipient addresses from QMGR, creates a recipient structure using

`smar_rcpt_new(smar_ctx, &smar_rcpt)`

fills in the necessary data, and invokes

`smar_rcpt_rslv(smar_ctx, smar_rcpt)`

which first checks the "mailertable" and if it can't find a matching entry, then it will invoke the DNS resolver (see Section 4.3.10):

`dns_req_add(dns_mgr_ctx, q, T_MX, smar_rcpt_cb, smar_rcpt)`

The callback function

`smar_rcpt_cb(dns_res_P dns_res, void *ctx)`

locks `smar_ctx` and analyzes `dns_res` as follows.

If the result is `DNSR_NOTFOUND` and the lookup was for an MX record then it will simply try to find A records. If it is another error then a general error handling section will be invoked.

First it will be checked whether the query was for an A record in which case `arr_c_mx` is incremented. Then the error type (temporary or permanent) is checked and a flag in `smar_rcpt` is set in the former case. An error is returned to the caller iff the query was for an MX record or the query was for an A record and all records have been received and there was a temporary error. If that is not the case, but all open requests are answered, then the results are returned to the caller using `smar_rcpt_re_all(smar_rcpt)` and thereafter the rcpt structure is freed using `smar_rcpt_free(smar_rcpt)`.

If the result was not an error then two cases have to be taken care of:

1. MX record: set the variable `arr_n_mx` to the number of MX records (1 if none has been found, thereby "pretending" that the host has exactly one MX record, i.e., itself), and initialize `arr_c_mx` and `arr_n_a` to 0. Allocate an array of the correct size (`arr_n_mx`) to hold the results and populate that array with the return values from the DNS query. Additionally start a DNS request for each hostname to get its A records: `dns_req_add(dns_mgr_ctx, q, T_A, smar_rcpt_cb, ctx)`.
2. A record: for each returned value look up the matching MX name in the result array `arr_res` and then store all A records in a newly allocated array `ardns_a`. If all entries have been received then the results are returned to the caller (see above).

4.8.2 Alias Expansion

Expanding aliases makes the address resolver significantly more complex. Unfortunately the current implementation does not allow for a simple layer around the current recipient resolver. This is due to the asynchronous nature of the DNS library which requires the use of callback functions. As explained in the previous section, the callback function checks whether all results arrived in which case it will put the data into an RCB and send it back to QMGR.

4.8.3 Aliases: Owner- Handling

Question: how to implement owner-alias and alias-request? Problem: bounces go to owner-alias (see also Section 2.6.7). Does this mean a transaction should be "cloned" or should the transaction context be extended? What if mail is sent to two lists and two "normal" rcpts?

```
mail from:<sender>
rcpt to:<list1>
rcpt to:<list2>
rcpt to:<rcpt1>
rcpt to:<rcpt2>
```

Usually a bounce goes to sender. However, if mail to listX fails, the function that creates bounces needs to handle this differently. Here's a list of possible ways to handle `owner-` aliases and the problems with them.

1. Proposal: keep a list of owner-alias in the transaction context and reference them by index in each recipient. Plus a flag saying that it's valid, or less than 0 for not valid? Or 0: original sender, greater than 0: index+1 (because the content is initialized to 0 so any other value must be explicitly set).

   Problems:

   (a) alias lists can be expanded recursively, so just keeping the original recipient addresses (listX) might not be enough, there can be (intermediate) addresses. It is still possible to handle this by having a list of those addresses in the transaction context. However, as soon as there is more data than just the address to handle, this is getting more complicated to implement, and harder to maintain. Examples:

      i. Some counters in the transaction context are per sender address, e.g., the number of recipients that have been sent to the address resolver. This counter is used by the scheduler to determine whether it can create a delivery transaction – if most of the recipients are still being resolved the MX piggybacking is inefficient. However, the counter must be per sender address because a delivery transaction is of course only for one sender address, hence a single counter is wrong if there are multiple owner-alias addresses. The counter is also wrong in case of bounces because those are sent to a different recipient (the original sender).

      ii. There might be DSN requests per recipient address. *Question*: how are these inherited by expanded recipients? In some case those atttributes most not be inherited, e.g., requests for delivery notifications, otherwise the members of the list would be exposed.

   (b) each recipient is expanded individually, which in turn means there can be multiple owner-aliases in the transaction context (one list per recipient). This can require that arrays are grown to accommodate the additional addresses, which is always a problem because the additional memory allocation can fail. Moreover, it makes record keeping more complicated.

2. Proposal: Maybe it would be useful to keep a "tree" of recipients? Mark a recipient as "expanded" (i.e., it was the LHS of an alias) and have a flag "has-owner". Then other recipients can refer to that recipient (by index).

   Problems:

   (a) how to deal with those "expanded" recipients? They must be removed when all the references are gone. In the worst case they can be removed when the transaction is removed. In that case the transaction context must have a list of expanded recipients (those could be stored in a bitfield), or the queue API (DEFEDB) must provide a function to remove all entries which refer to a transaction id.

(b) When recipients are read from DEFEDB, the corresponding expanded recipients must be read too, otherwise the address to use as envelope sender is missing. This increases the number of items in AQ.

3. Proposal: "envelope splitting" as used in sendmail 8 "solves" the problems of proposal 2 by creating new transactions. This avoids the handling of special recipients that contains not much more information than just the owner address.

   Problems:

   (a) It requires to create new transaction identifiers which is currently only done by the SMTP servers. It might be possible to give QMGR a reserved SMTP server "process id" which can be used to generate unique transaction identifiers, see Section 3.5.2.4.2.

   (b) Envelope splitting causes multiple transactions to refer to the same content. This makes deciding when to unlink an entry in CDB significantly more complicated. sendmail 8 relies on the link count of the filesystem (check), but sendmail X should not do that because it makes CDB more complicated and dependent on a certain implementation (unless emulated for others which is not easy because CDB is currently just a library, not a process that maintains data).

   It might be possible to add another counter: `aqt_clones` which counts the number of cloned transactions. A cloned transaction contains a link to the original transaction (just the TA id). If the number of recipients `aqt_rcpts_left` is decreased to zero, then it is checked whether the TA is cloned in which case the clone counter in the original TA is decreased. A CDB entry is removed if the two counters `aqt_rcpts_left` and `aqt_clones` in a original TA are both zero.

Overall, proposal 3 seems like the "cleanest" solution even though it currently (2004-06-30) has the biggest impact on the implementation, i.e., it requires a lot of changes.

4.9 Milter

4.9.1 Sendmail 8 Milter Protocol

Note: this is based on an old document and might not be up to date; it is listed here only as a starting point for the sendmail X protocol.

The protocol consists of simple data strings which are tagged with a single character. The first element of each exchange is a 32 bit integer which denotes the total length of the string to follow, including the command and maybe `NUL` characters. Note: since we control both sides, we can rely on those. If the data is incorrect, we can simply terminate the connection and log an error.

The characters for specifying the data are:

MTA to Milter (upper case char):

char	meaning	parameters
A	Abort	-
B	Body chunk	buffer `NUL`
C	Connection information	hostname `NUL` `family` `port` `sockinfo`
D	Define macro	`Where`[6] (name `NUL` value `NUL`) *
E	final body chunk (End)	buffer `NUL`
H	HELO	heloname `NUL`
L	Header	field `NUL` value `NUL`
M	MAIL from	sender `NUL` (arg `NUL`)*
N	End of headers (EOH)	-
O	Option negotiation	`version` `fflags` `pflags`
Q	QUIT (once per connection)	-
R	RCPT to	recipient `NUL` (arg `NUL`)*

The buffers that transfer body chunks can contain `NUL` characters, hence the length parameter is important. For `fflags` and `pflags` see 4.9.1.2.

`version`, `fflags`, and `pflags` are 32 bit integers (network format). `family` is one byte:

`family`	data
`U`	unknown; SOCKADDR = `NULL`
4	AF_INET
6	AF_INET6
L	AF_UNIX

`port` is a 16 bit integer (currently assumed to be a `u_short` – need `uint16_t` to assure this) (network format) (empty for `U`).

`family`	`sockinfo`
`U`	empty
4	string (inet_aton/ntoa)
6	string (inet_pton/ntop) (optional IPv6: address prefix)
`L`	path

unless `U` is used, the string includes the trailing `NUL`.

Milter to MTA (lower case and punctuation char, except for negotiation):

char	meaning	parameters
O	Option negotiation	`version` `fflags` `pflags`
+	add recipient	rcpt `NUL`
-	remove recipient	rcpt `NUL`
a	accept	-
b	replace body (chunk)	buffer `NUL`
c	continue	-
d	discard	-
h	add header	field `NUL` value `NUL`
m	modify header	index (32 bit) field `NUL` replacement `NUL`
p	progress	-
q	quarantine	reason `NUL`
r	reject	-
t	tempfail	-
y	reply code	rcode SPACE xcode SPACE message `NUL`

4.9.1.1 Macros

Macros and their values can precede Connect, HELO, MAIL, and RCPT. The first character (`Where`) after the command denotes the command character to which those macros belong (`C`, `H`, `M`, `R`). Implementation note: the entire "string" is stored in the context (array of pointers), and searched in `smfi_getsymval()`.

The default macros expected to be passed from MTA to milter per-command are (this can be changed using the `Milter.macros.*` options):

`Where`	Macros
`C`	daemon_name, if_name, if_addr, j, _
`H`	tls_version, cipher, cipher_bits, cert_subject, cert_issuer
`M`	i, auth_type, auth_authen, auth_ssf, auth_author, mail_mailer, mail_host, mail_addr
`R`	rcpt_mailer, rcpt_host, rcpt_addr

All macros stay in effect from the point they are received until the end of the connection with the exception of rcpt_* which should only be set during the call to `xxfi_envrcpt()`.

4.9.1.2 Option Negotiation

1. MTA sends libmilter SMFI_VERSION, SMFI_CURR_ACTS, SMFI_CURR_PROT (`version`, `fflags`: possible actions, `pflags`: possible extra commands)
2. libmilter aborts if MTA version is less than filter version
3. libmilter aborts if MTA flags don't cover all requested filter flags
4. libmilter aborts if MTA protocol bits don't cover `ctx_pflags` (protocol related)
5. libmilter sends MTA filter version (`xxfi_version`), filter flags (`xxfi_flags`: required actions), and protocol bits (`ctx_pflags`).
6. MTA aborts if libmilter version is 1 or greater than MTA version (SMFI_VERSION)
7. MTA aborts if libmilter flags aren't all included in MTA flags (SMFI_CURR_ACTS)
8. MTA aborts if libmilter protocol bits aren't all included in MTA protocol bits (SMFI_CURR_PROT)
9. Work continues as normal.

Note: there should be a better way to deal with differences in the available actions and extra commands: capabilities and requirements should be announced and then each side decides on its own whether those are sufficient and fulfilled, respectively.

4.9.2 SMTPS - Milter Protocol

Here are the protocol exchange types:

RT_S2M_NID	uint32	id of SMTPS, new connection
RT_S2M_ID	uint32	id of SMTPS
RT_S2M_CID	uint32	id of SMTPS, closing (shutting down)
RT_S2M_MAXTHRDS	uint32	max number of threads
RT_S2M_NSEID	str	new session id
RT_S2M_EHLO	str	ehlo string
RT_S2M_SEID	str	session id
RT_S2M_CSEID	str	close session id
RT_S2M_CLTIP4	uint32	client IPv4 address
RT_S2M_NTAID	str	new transaction id
RT_S2M_TAID	str	transaction id
RT_S2M_MAIL	str	MAIL from
RT_S2M_RCPT_IDX	uint32	rcpt idx
RT_S2M_RCPT	str	RCPT to
RT_S2M_DATA	str	data
RT_S2M_BODY	str	body chunk
RT_S2M_DOT	str	final data chunk
RT_S2M_ARG	str	argument (for MAIL, RCPT)
RT_S2M_MACW	uint32	macro: `Where` (see 4.9.1.1)
RT_S2M_MACN	str	macro name
RT_S2M_MACV	str	macro value
RT_S2M_PROT	uint32	SMTPS - PMILTER protocol version
RT_S2M_CAP	uint32	SMTPS capabilities
RT_S2M_REQ	uint32	SMTPS requirements

Initial setup: SMTPS sends libmilter its id (RT_S2M_NID) and the maximum number of threads (RT_S2M_MAXTHRDS) it is going to create. The latter value can be used by libmilter to size internal tables if necessary.

ToDo: option negotation: must be better than what is done in sm8 (see 4.9.1.2), see 4.9.2.1 for possible solutions.

For the first test implementation it is: SMTP server sends RT_S2M_PROT, RT_S2M_CAP, RT_S2M_REQ, RT_S2M_MAXTHRDS. Milter responds with RT_M2S_ID, RT_M2S_PROT, RT_M2S_CAP, RT_M2S_REQ,

The rest follows ESMTP (all protocol exchanges include RT_S2M_ID as first element).

1. connection from client: RT_S2M_NSEID, RT_S2M_CLTIP4 (later on it could be more, see sm8 version).

2. EHLO command received: RT_S2M_SEID, RT_S2M_EHLO

3. MAIL command has been given: RT_S2M_SEID, RT_S2M_NTAID, RT_S2M_MAIL, if necessary: RT_S2M_ARG (maybe multiple times).

4. RCPT command has been given. RT_S2M_SEID, RT_S2M_TAID, RT_S2M_RCPT_IDX, RT_S2M_RCPT if necessary: RT_S2M_ARG (maybe multiple times).

5. DATA command has been given. RT_S2M_SEID, RT_S2M_TAID, RT_S2M_DATA

6. body chunks RT_S2M_SEID, RT_S2M_TAID, RT_S2M_BODY

7. Final dot of DATA section has been given RT_S2M_SEID, RT_S2M_TAID, RT_S2M_DOT

8. At the appropriate stages in the protocol macros can be transferred: RT_S2M_MACW (one character), RT_S2M_MACN, RT_S2M_MACV.

9. QUIT: RT_S2M_CSEID.

Replies use the following record types:

RT_M2S_ID	uint32	id of SMTPS
RT_M2S_SEID	str	session id
RT_M2S_RCODE	uint32	SMTP reply code
RT_M2S_PROT	uint32	SMTPS - PMILTER protocol version
RT_M2S_CAP	uint32	PMILTER capabilities
RT_M2S_REQ	uint32	PMILTER requirements

Each reply must contain the SMTP server id RT_M2S_ID and the session id RT_M2S_SEID.

The possible reply codes are:

250	accept
350	continue
450	temporary error
550	permanent error

As specified in the sendmail 8 libmilter documentation "accept" means that the entire session or transaction will be accepted and no further commands should be sent to the milter. "continue" means the milter wants session/transaction to continue, while the other two have the obvious meaning.

If an SMTP server shuts down, it notifies libmilter by sending RT_S2M_CID.

4.9.2.1 Option Negotiation

Basic idea:

protocol version: major/minor, major must match, minor can be different.

Announce capabilities and requirements (both sides). libmilter: a milter can register a callback the decides whether the capabilities (of libmilter and MTA: libmilter can be dynamically linked, or some source code is linked against an *old* version) are sufficient and give its OK (or error and hence abort).

4.9.2.2 Differences to Milter

The most obvious difference is that a policy milter cannot change the body of a mail nor can it add or replace headers. It also cannot delete or add recipients.

Another difference is that sendmail X does not send headers individually to the policy milter, it just send the mail (in chunks). It is probably the best to provide a library function that can parse headers which can be called by a milter if necessary. It would maintain a state (how far it read in the buffer). A small problem may arise if the headers are larger than the first buffer because then it might need to reassemble a header line (if it crosses a buffer boundary which would be very likely if it actually exceeds the size).

4.10 Implementation of Databases and Caches for Envelopes, Contents, and Connections

4.10.1 Incoming Envelope Database

IQDB is currently implemented as typed RSC, see Section 4.3.5.1. IQDB simply contains references to qss_sess_T, qss_ta_T, and qss_rcpts_T, see Section 4.7.3, it does not have its own data structures, it merely allows for a way to access the data via a key (SMTPS session, transaction, recipient identifier).

4.10.2 Incoming Envelope Database Disk Backup

The API of IBDB is described in Section 3.13.4.3.

The current implementation requires more functions than described there. This is an outcome of the requirement to updata the various queues safely and in a transaction based manner. To achieve this, a list of change request can be maintained; the elements of this list have the following structure:

```
struct ibdb_req_S
{
  unsigned int              ibdb_req_type;
  int                       ibdb_req_status;
  sessta_id_T               ibdb_req_ss_ta_id; /* SMTPS transaction id */
  sm_str_P                  ibdb_req_addr_pa;  /* MAIL/RCPT address */
  cdb_id_P                  ibdb_req_cdb_id;
  unsigned int              ibdb_req_nrcpts;
  rcpt_idx_T                ibdb_req_rcpt_idx; /* RCPT index */
  SIMPLEQ_ENTRY(ibdb_req_S) ibdb_req_link;
};
```

The functions to deal with the request lists are:

`sm_ret_T ibdb_rcpt_app(ibdb_ctx_P ibdb_ctx, ibdb_rcpt_P ibdb_rcpt, ibdb_req_hd_P ibdb_req_hd, int status)` appends a recipient change request.

`sm_ret_T ibdb_ta_app(ibdb_ctx_P ibdb_ctx, ibdb_ta_P ibdb_ta, ibdb_req_hd_P ibdb_req_hd, int status)` appends a transaction change request.

`sm_ret_T ibdb_req_cancel(ibdb_ctx_P ibdb_ctx, ibdb_req_hd_P ibdb_req_hd)` cancel all the requests in the list.

`sm_ret_T ibdb_wr_status(ibdb_ctx_P ibdb_ctx, ibdb_req_hd_P ibdb_req_hd)` perform the status updates as specified by the request list.

4.10.3 Active Envelope Database

The active queue currently uses the following structure as context:

```
struct aq_ctx_S
{
```

```
  pthread_mutex_t  aq_mutex;      /* only one mutex for now */
  unsigned int     aq_limit;      /* maximum number of entries */
  unsigned int     aq_entries;    /* current number of entries */
  unsigned int     aq_t_da;       /* entries being delivered */

  aq_tas_T         aq_tas;
  aq_rcpts_T       aq_rcpts;
};
```

For now we just use lists of aq_ta/aq_dta/aq_rcpt structures. Of course we need better access methods later on. Currently we will use FIFO as only scheduling strategy.

A recipient context consists of the following elements:

```
struct aq_rcpt_S
{
  sessta_id_T              aqr_ss_ta_id;  /* ta id in SMTPS */
  sessta_id_T              aqr_da_ta_id;  /* ta id in DA */
  sm_str_P                 aqr_rcpt_pa;   /* printable addr */
  smtp_status_T            aqr_status;    /* status */
  smtp_status_T            aqr_status_new;        /* new status */
  unsigned int             aqr_err_st;    /* state which caused error */
  unsigned int             aqr_flags;     /* flags */
  rcpt_idx_T               aqr_rcpt_idx;  /* rcpt idx */
  unsigned int             aqr_tries;     /* # of delivery attempts */
  unsigned int             aqr_da_idx;    /* DA idx (kind of DA) */

  /* SMTPC id (actually selected DA) */
  int                      aqr_qsc_id;

  /*
  **  HACK! Need list of addresses. Do we only need IP addresses
  **  or do we need more (MX records, TTLs)? We need at least some
  **  kind of ordering, i.e., the priority. This is needed for the
  **  scheduler (if a domain has several MX records with the same
  **  priority, we can deliver to any of those, there's no order
  **  between them). Moreover, if we store this data in DEFEDB,
  **  we also need TTLs.
  */

  /*
  **  Number of entries in address array.
  **  Should this be "int" instead and denote the maximum index,
  **  where -1 means: no entries?
  **  Currently the check for "is there another entry" is
  **      (aqr_addr_cur < aqr_addr_max - 1)
  **  i.e., valid entries are 0 to aqr_addr_max - 1.
  */

  unsigned int             aqr_addr_max;
  unsigned int             aqr_addr_cur;  /* cur idx in address array */
```

```
aq_raddr_T             *aqr_addrs;     /* array of addresses */

/* XXX Hack */
ipv4_T                  aqr_addr_fail; /* failed address */

/* address storage to use if memory allocaction failed */
aq_raddr_T              aqr_addr_mf;

time_T                  aqr_entered;   /* entered into AQ */
time_T                  aqr_st_time;   /* start time (rcvd) */
time_T                  aqr_last_try;  /* last time scheduled */

/* next time to try (after it has been stored in DEFEDB) */
time_T                  aqr_next_try;

/* Error message if delivery failed */
sm_str_P                aqr_msg;

/*
**  Bounce recipient: stores list of recipient indices for which
**  this is a bounce message.
**  Note: if this is stored in DEFEDB, then the array doesn't need
**  to be saved provided that the recipients are removed in the
**  same (DB) transaction because the bounce recipient contains
**  all necessary data for the DSN. If, however, the recipients
**  are not removed "simultaneously, then it is a bid harder to
**  get consistency because it isn't obvious for which recipients
**  this bounce has been created. That data is only indirectly
**  available through aqr_bounce_idx (see below).
*/

sm_str_P                aqr_dsn_msg;
unsigned int            aqr_dsn_rcpts;
unsigned int            aqr_dsn_rcpts_max;
rcpt_idx_T             *aqr_dsns;      /* array of rcpt indices */

/*
**  rcpt idx for bounce: stores the rcpt_idx (> 0) if a bounce
**  for this recipient is generated and being delivered.
**  This is used as "semaphore" to avoid multiple bounces for
**  the same recipient (needs to be stored in DEFEDB).
*/

rcpt_idx_T              aqr_bounce_idx;

/* linked list for AQ, currently this is the way to access all rcpts */
TAILQ_ENTRY(aq_rcpt_S)  aqr_db_link;   /* links */

/*
**  Linked lists for:
**  - SMTPS transaction:
```

```
  **      to find all recipients for the original transaction
  **      (to find out whether they can be delivered in the same
  **      transaction, i.e., same DA, + MX piggybacking)
  **  - DA transaction:
  **      to find the recipients that belong to one delivery attempt
  **      and update their status
  **  Link to ta:
  **      to update the recipient counter(s).
  */

  sm_ring_T                aqr_ss_link;
  sm_ring_T                aqr_da_link;
  aq_ta_P                  aqr_ss_ta;     /* transaction */
};
```

This structure contains linked lists for:

- for the active queue itself `aqr_db_link`, currently this is the way to access all recipients
- SMTPS transaction `aqr_ss_link`: to find all recipients for the original transaction (to find out whether they can be delivered in the same transaction, i.e., same DA, and for MX piggybacking).
- DA transaction `aqr_da_link`: to find the recipients that belong to one delivery attempt and update their status.

A reference to the transaction `aqr_ss_ta` is used to make it easier to update the recipient counter(s).

A transaction context has these elements:

```
struct aq_ta_S
{
  /* XXX other times? */
  time_T          aqt_st_time;           /* start time (received) */
  aq_mail_P       aqt_mail;              /* mail from */
/* XXX only in aq_da_ta */
  unsigned int    aqt_rcpts;     /* number of recipients in AQ */
  unsigned int    aqt_rcpts_ar;  /* rcpts to receive from AR */
  unsigned int    aqt_rcpts_arf; /* #of entries with SMAR failure */

  /* Number of recipients in DEFEDB */
  unsigned int    aqt_rcpts_tot; /* total number of recipients */
  unsigned int    aqt_rcpts_left;        /* rcpts still to deliver */
  unsigned int    aqt_rcpts_temp;        /* rcpts temp failed */
  unsigned int    aqt_rcpts_perm;        /* rcpts perm failed */
  unsigned int    aqt_rcpts_tried;       /* rcpts already tried */
  rcpt_idx_T      aqt_nxt_idx;   /* next recipient index */

  unsigned int    aqt_state;
  unsigned int    aqt_flags;

  /*
```

```
  **  rcpt idx for (double) bounce; when a bounce is needed a recipient
  **  struct is created, its rcpt_idx is this bounce_idx.
  **  It should be aqt_rcpts_tot (+1) when it is created; afterwards
  **  aqt_rcpts_tot is increased of course.
  */

  rcpt_idx_T       aqt_bounce_idx;
  rcpt_idx_T       aqt_dbl_bounce_idx;

  sessta_id_T      aqt_ss_ta_id;  /* ta id in SMTPS */
  cdb_id_P         aqt_cdb_id;
  TAILQ_ENTRY(aq_ta_S)            aqt_ta_l;       /* links */

  /* XXX add list of recipients? that makes lookups easier... see above */
};
```

The field `aqt_ta_l` links all transactions in the active queue together.

As it can be seen, there are many counters in the transaction context:

- Two counters are related to the recipients that are currently in the active queue, which may be less than the number of recipients of the transaction.
  1. `aqt_rcpts`: number of recipients in AQ.
  2. `aqt_rcpts_ar`: number of recipients still to receive from AR, i.e., their routing information is currently determined.
- These counters are related to the overall number of recipients, no matter whether they are in the active queue or elsewhere (DEFEDB, IBDB):
  1. `aqt_rcpts_tot`: total number of recipients.
  2. `aqt_rcpts_left`: recipients still to deliver.
  3. `aqt_rcpts_perm`: recipients permanently failed.

  The number of recipients that encountered a temporary failure `aqt_rcpts_temp` does not reflect those recipients that are in DEFEDB. When a recipient is written to DEFEDB, the status flag indicating a temporary failure will not be saved. Hence when a recipient is tried again, the previous temporary failure is mostly ignored except for bookkeeping.

4.10.4 Deferred Envelope Database Implementation

The current implementation (2003-01-01) of the main queue uses Berkeley DB 4.1 (proposal 2 from Section 3.13.6.1).

The main context looks like this:

```
typedef SIMPLEQ_HEAD(, edb_req_S)       edb_req_hd_T, *edb_req_hd_P;
struct edb_ctx_S
{
  pthread_mutex_t  edb_mutex;     /* only one mutex for now */
```

```
  unsigned int     edb_entries;   /* current number of entries */
  edb_req_hd_T     edb_reql_wr;   /* request list (wr) */
  edb_req_hd_T     edb_reql_pool;
  DB_ENV          *edb_bdbenv;
  DB              *edb_bdb;       /* Berkeley DB */
};
```

A change request has the following structure:

```
struct edb_req_S
{
  unsigned int                  edb_req_type;
  smtp_id_T                     edb_req_id;
  sm_rcb_P                      edb_rcb;
  SIMPLEQ_ENTRY(edb_req_S)      edb_req_link;
};
```

The context maintains two lists of requests: a pool of free request list entries (`edb_reql_pool`) for reuse, and a list of change requests (`edb_reql_wr`) that are committed to disk when `edb_wr_status(edb_ctx_P edb_ctx)` is called.

A request itself contains a type (TA or RCPT), and identifier (which is used as key) and a RCB that stores the appropriate context in encoding form as defined by the RCB format. The type can actually be more than just transaction or recipient, it can also denote that the entry matching the identifier should be removed from the DB.

The data structures for transactions (mail sender) and recipients are shared with the active queue, see Section 4.10.3.

See Section 3.13.6 for the DEFEDB API, here's the current implementation:

Two functions are available to open and close a DEFEDB:

- `sm_ret_T edb_open(edb_ctx_P *edb, unsigned int size)`: open a DEFEDB, `size` is an estimate for the size (currently unused).
- `sm_ret_T edb_close(edb_ctx_P edb)`: close a DEFEDB.

The functions which take a parameter `edb_req_hd_P edb_req_hd` will use that argument as the head of the request list unless it is `NULL`, in which case the write request list `edb_reql_wr` of the context will be used.

- `sm_ret_T edb_rcpt_app(edb_ctx_P edb_ctx, aq_rcpt_P aq_rcpt, edb_req_hd_P edb_req_hd, int status)`: append a recipient context to the request list, i.e., encode the recipient context into an RCB and add an entry to the request list.
- `sm_ret_T edb_ta_app(edb_ctx_P edb_ctx, aq_ta_P aq_ta, edb_req_hd_P edb_req_hd, int status)`: append a transaction context to the request list, i.e., encode the transaction context into an RCB and add an entry to the request list.
- `sm_ret_T edb_wr_status(edb_ctx_P edb_ctx, edb_req_hd_P edb_req_hd)`: write request list to disk. This is done as one DB transaction; only if the transaction succeeded the request list is cleared.

To retrieve an entry from the DEFEDB one function is provided:

- `sm_ret_T edb_rd_req(edb_ctx_P edb_ctx, edb_req_P edb_req)`: `edb_req` must be filled with the type and the appropriate identifier (recipient or transaction), which is used to lookup the matching entry in the DB. If it is found, the RCB in the request is filled in.

To decode the RCB retrieved via `edb_rd_req()` and fill out an active queue context of the correct type the following two functions are available:

- `sm_ret_T edb_rcpt_dec(edb_req_P edb_req, aq_rcpt_P aq_rcpt)`
- `sm_ret_T edb_ta_dec(edb_req_P edb_req, aq_ta_P aq_ta)`.

To read through a DEFEDB these four functions are provided:

- `sm_ret_T edb_rd_open(edb_ctx_P edb_ctx, edb_cursor_P *pedb_cursor)`: prepare the DB for reading (open a cursor).
- `sm_ret_T edb_rd_next(edb_ctx_P edb_ctx, edb_cursor_P edb_cursor, edb_req_P edb_req)`: get the next entry.
- `sm_ret_T edb_get_type(edb_req_P edb_req)`: get the type of the entry just read (TA or RCPT).
- `sm_ret_T edb_rd_close(edb_ctx_P edb_ctx, edb_cursor_P edb_cursor)`: close the cursor after reading entries.

To remove a transaction or a recipient from the DB (directly) use:

- `sm_ret_T edb_ta_rm(edb_ctx_P edb_ctx, sessta_id_T ta_id)`
- `sm_ret_T edb_rcpt_rm(edb_ctx_P edb_ctx, rcpt_id_T rcpt_id)`.

To add a request to remove a transaction or a recipient from the DB use:

- `sm_ret_T edb_ta_rm_req(edb_ctx_P edb_ctx, sessta_id_T ta_id, edb_req_hd_P edb_req_hd)`
- `sm_ret_T edb_rcpt_rm_req(edb_ctx_P edb_ctx, rcpt_id_T rcpt_id, edb_req_hd_P edb_req_hd)`

and commit that request later on with `edb_wr_status()`.

Internal functions to manage a request entry or list are:

- `sm_ret_T edb_req_new(edb_ctx_P edb_ctx, edb_req_P *pedb_req)`: get a new request structure (from the pool or allocate a new one if the pool is empty).
- `sm_ret_T edb_reql_free(edb_req_hd_P _edb_req_hd)`: free an entire request list.

4.11 Misc

This section describes the implementation of various programs.

4.11.1 IBDB Cleanup

How to efficiently perform IBDB cleanup?

Try to minimize the amount of data to clean up. This can be done by performing rollovers at an appropriate moment, i.e., when the number of outstanding transactions and recipients is zero. This is probably only possible for low-volume sites. If those two values are zero, then all preceeding files can be removed.

Read an IBDB file and create a new one that has only the open transactions/recipients in there? Leave "holes" in the sequence, e.g., use 0x1-0xf and leave 0 free for "cleaning", i.e., read 0x1-0xf and then write all the open transactions into 0. Problem: what to do with repeated passes?

How about different names (extensions) instead?

It might be possible to ignore logfiles that are older than the transactional timeout. Those logfiles can only contain data about transaction that have been either completed or have timed out. Neither of these are of interest for the reconstruction of the queue. Does this mean a very simple cleanup process is possible which simply removes old logfiles? This minimizes the amount of effort during runtime at the expense of diskspace and startup after an unclean shutdown. For the first sendmail X version this might be a "good enough" solution.

4.11.2 Show Mail Queue

Section 3.11.1 posed the question: how accurate must the output be? Here is a closer look at the problem: using Berkeley DB for the implementation of the deferred queue poses some problems because there is no "read-only" access to the content, at least not in the current implementation: the function to open a DB environment offers no read-only option. Obviously it is a bad idea to open the environment read/write just to read the queue: the program should not make any modifications to the DB; sendmail X is designed such that there is a single writer for it only, all locking is done inside QMGR, nothing relies on locking within Berkeley DB (this is done to avoid the DB corruption problems that may occur otherwise, especially if some other program crashes while accessing the DB). However, there is a trick that gives us read-only access: open the DB directly without opening the environment. In this case we can make the DB group readable, which should be used to avoid any type of corruption. Even though this looks like a nice solution to the main problem, it opens up another one: the DB may not contain up-to-date information. Berkeley DB keeps a log file (if used in transaction mode) and hence the DB itself might have old (and even inconsistent) data.

An alternative approach is to query QMGR about the content of the deferred queue. This allows us to keep our single writer (and reader), but it causes other problems, e.g., locking the DB for quite some time and hence causing performance problems.

Note: a fairly accurate state of the mail queue content can be gathered from CDB and the logfile: the CDB file names are the SMTP server transaction ids which can be looked up in the logfile to find the sender and recipients. To find out which recipients have not yet been delivered, the logfile must be scanned for that data. This can be time consuming but it is less intrusive.

4.12 Testing

As requested in Section 1.1.3.5.1 there are many test programs which can as usual be invoked by `make check`.

Some of them are included in the directories for which they perform tests, e.g., `smar` and `libdns`, but most of them are in the directories `checks`, `check2`, and `chkmts`. There are several "check" directories because of some length restrictions on AIX (2048 bytes line length for some utilities used by the `autoconf` tools).

The test programs range from very simple: testing a few library functions ("unit tests"), e.g., string related, to those which test (aspects of) the entire MTS.

4.13 Problems encountered in the Implementation

This section describes some of the problems that have been encountered in the implementation of sendmail X. There are two different kind of problems:

1. finding the right implementation for algorithms that satisfy the requirements of sendmail X;
2. problems in the behavior of sendmail X due to the implementation.

4.13.1 Implementation Problems

Implementing sendmail X as a few modules that are required to run under (almost) all circumstances turned out to be significantly more complicated than expected.

4.13.1.1 Out of Memory

The most complicated problem is to deal with out-of-memory errors. Note: proper recovery from this is still an open problem. This is even more so a problem because some OS allow to "overcommit" memory. As soon as overcommitted memory is actually used and the need can not be satisfied, the OS "randomly" kills a process to allow progress of the entire system. Even though the result of every memory allocation call in sendmail X is carefully checked and error handlers are invoked if the call returns `NULL`, these efforts may be useless on those OSs.

Dealing with memory shortage is also severely hampered by the non-availability of a (standard) function to actually determine the current memory usage of a process. The function `getrusage(2)` returns a structure that has some fields which can contain information about the memory usage, but some OSs do not put valid data into those fields, and there is only one field (`ru_maxrss`) that is related to memory usage[7]. It might be possible that the memory allocation functions try to keep track of the amount of used memory themselves, but that requires that the `free()` function receives an additional argument which denotes the size of the memory segment, otherwise the memory allocation module needs to keep track of that information itself which is a waste of memory and programming effort. Moreover, due to internal fragmentation the value might not be correct anyway.

4.13.1.2 Transaction Based Processing

Another problem is transaction based processing, see also Section 4.1.3. It turns out to be non-trivial to implement this, i.e., a function must either successfully complete all of its operations or none of them, especially if multiple functions are involved in a sequence, because this means that those functions need to have some kind of "undo" counterparts which can be invoked in case of errors.

[7] Others use some "integral" value of memory over time.

4.13.2 Behavior of the Implementation

4.13.2.1 Number of Open Outgoing Connections

The communication between the various modules of sendmail X turns out to be a problem due to the latency. The most significant example of this is the interaction of the scheduler and the delivery agents. The scheduler needs to keep track of the number of open (outgoing) connections to implement the slow start algorithm and to obey limits set by the user. However, some tests show that the actual number of open connections to a client and the data that the scheduler has can be significantly different. For example, in some tests the scheduler reached an upper limit of 256 open connections while the sink program only listed 2 open connections, while SMTPC itself had some (low) number of busy threads, e.g., 4 to 16. There are two different problems:

1. The “real” number of open connections versus the number of busy threads in SMTPC.
2. The number of busy threads in SMTPC versus the number of scheduled sessions in QMGR.

One reason for these disagreements is that the messages are very small, hence it takes a very short time to deliver them (in a lab environment). This can be demonstrated by using larger messages sizes (the default is less than 1KB): for a message size mix of 32KB and 64KB the number of concurrent connection reaches the limit in the sink.

About item 2: Here is what seems to happen: as long as the concurrency limit (in qmgr) is not reached, the scheduler sends task to SMTPC; it stops only when the limit is reached or no more entries are in AQ (for scheduling). The scheduler has a lock on AQ hence no updates from SMTPC can be done. When the scheduler releases the lock results from SMTPC can be read. The scheduler is activated as soon as a result comes back from SMTPC, it then immediately schedules more entries (just one because the limit is reached) and the ping-pong (one result, one entry scheduled) continues.

Chapter 5

Sendmail X: Performance Tests and Results

5.1 SMTP Server Daemon

Remark (placed here so it doesn't get lost): there is a restricted number (< 60000) of possible open connections to one port. Could that limit the throughput we are trying to achieve or is such a high number of connections unfeasible?

5.1.1 SMTP Sink

For simple performance comparisons several SMTP sinks have been implemented or tested.

Test programs are:

1. smtp-sink from postfix. This is an entirely event driven program.
2. thrperconn: one thread per connection.
3. thrpool: uses a worker model with concurrency limiting, see Section 3.20.4.1.
4. smtps: state-threads, see Section 3.20.3.1.

Test machines are:

1. v-sun: Sun SPARCserver E450, 4 processors
2. v-bsd: FreeBSD 3.4, 2 PIII processors, 2 GB RAM
3. v-aix: AIX, 4 processors
4. schmidt: Linux 2.4, uses 15 threads per client only, otherwise the machine just "dies".

Entries in the tables down below denote execution time in seconds unless otherwise noted, hence smaller values are better.

Tests have been performed with myslam (a multi-threaded SMTP client), using 7 to 8 client machines, 50 threads per client, and 5000 messages per client.

1. v-sun (8 clients):

parameters	smtp-sink	smtps	thrperconn	thrpool
1KB/msg (40MB)	45s	70s	92s	43s
4KB/msg (160MB)	49s	56s	259s	78s
32KB/msg (1280MB)	203s	208s	>999s	110s
-w 1	141s	109s	156s	230s

Note: v-sun is a four processor machine, hence the multi-threaded programs (thrpool, thrperconn) can use multiple processors. I didn't select (via an option) multiple processors for smtps though.

Just as one example, the achieved throughput in MB/s is listed in the next table. As it can be seen, it is an order of magnitude lower than the sustainable throughput that can be achieved over a single connection (about 85-90MB/s measured with ttcp; this is a 100Mbit/s ethernet).

parameters	smtp-sink	smtps	thrperconn	thrpool
1KB/msg (40MB)	0.9	0.6	0.4	0.9
4KB/msg (160MB)	3.3	2.9	0.6	2.1
32KB/msg (1280MB)	6.5	6.3	-	11.9

2. v-bsd:

parameters	smtp-sink	smtps	thrperconn	thrpool
1KB msg size	97	87	380	140
4KB msg size	108	130	1150	156
32KB msg size	208	197	fails	330
-w 1	165	138	484	223

3. v-aix:

parameters	smtp-sink	smtps	thrperconn	thrpool
1KB msg size	38	28	-	31
4KB msg size	34	33	-	31
32KB msg size	125	125	-	125
-w 1	125	125	-	155
				125 for 250/3

4. schmidt:

parameters	smtp-sink	smtps	thrperconn	thrpool
1KB msg size	45	44	165	74
4KB msg size	54	45	418	75
32KB msg size	217	167	fails	256
-w 1	370	360	-	337

5.1.1.1 SMTP Sink with CDB

2004-03-02

statethreads/examples/smtps3

5.1.1.1.1 wiz See Section 5.2.1.1, machine 1

```
wiz$ time ./smtpc2 -fa@b.c -Rx@y.z -t 100 -s 1000 -r localhost
```

sink program	FS	times (s)
smtps3	-	5
smtpss	UFS	17, 18
smtps3 -C	UFS	16, 17, 19

5.1.1.1.2 perf-lab source: s-6.perf-lab

sink: v-bsd.perf-lab

with -C

```
s-6.perf-lab$ time ./smtpc2 -t 100 -s 1000 -r v-bsd.perf-lab
   19.17s real     1.08s user     0.64s system
```

without -C

```
s-6.perf-lab$ time ./smtpc2 -t 100 -s 1000 -r v-bsd.perf-lab
    3.04s real     0.81s user     0.59s system
```

source: s-6.perf-lab

sink: mon.perf-lab (FreeBSD 4.9)

with -C

```
   12.05s real     1.04s user     0.67s system
```

without -C

```
    3.03s real     0.92s user     0.54s system
```

2004-03-04 source: s-6.perf-lab; sink: v-sun.perf-lab

with -C: 20s - 24s (UFS) Note: It takes 20s(!) to remove all CDB files:

```
time rm ?/S*     0m20.11s
```

with -C: 1s (TMPFS); 16s (UFS, /), rm: 14s; logging turned on: 16s, rm: 0.8s.

without -C: 1s

2004-03-08 source: s-6.perf-lab; sink: v-bsd;

./smtpc -t 100 -s 1000

sink program	time (s)
smtpss	30
smtps3 -C	30
smtps3	3

2004-03-08 source: s-6.perf-lab; sink: v-sun;

./smtpc -t 100 -s 1000

sink program	FS	times (s)
smtps3	-	1
smtpss	UFS	25, 30
smtps3 -C	UFS	23
smtpss	swap	2, 3
smtps3 -C	swap	1, 2

Note: the variance for `smtpss` on UFS is fairly large. The lower numbers are achieved by running `smtps3 -C` first and then `smtpss`, the larger numbers are measured when the CDB files have just been removed. However, this effect was not reproduceable. Note: removing those files takes about as long as a test run.

5.1.2 SMTP Relaying Using a Sendmail X Prototype

Test setup with a sendmail X prototype of 2002-09-04: v-aix.perf-lab running QMGR, SMTPS, and SMTPC. Relaying from localhost to v-bsd.perf-lab. Source program running on v-aix:

`time ./smtp-source -s 50 -m 100 -c localhost:8000`

Using the full version: 2.45s; turning fsync() off: 1.44s.

This clearly shows the need for a better CDB implementation, at least on AIX.

Same test with reversed roles (smX on v-bsd, sink on v-aix): using the full version: 7.44s; turning fsync() off: 6.20s. For comparison: using sendmail 8.12: 14.71s.

The SCSI disks on v-bsd seem to be fairly slow. Moreover, there seems to be something wrong with the OS version (it's very old: FreeBSD 3.4).

On FreeBSD 4.6 (machine 14, see Section 5.2.1.1) (source, sink, sm-9 of 2002-10-01 on the same machine):

`time ./smtp-source -s 100 -m 200 -c localhost:8000`

softupdates: 4.35s; without softupdates: 5.66s

`time ./smtp-source -s 50 -m 100 -c localhost:8000`

softupdates: 2.01s/1.93s, -U: 1.79s; without softupdates: 2.60s/2.46s, -U: 2.17s

(-U turns off fsync()).

Using sendmail 8.12.6:

`time ./smtp-source -s 50 -m 100 localhost:1234`

softupdates: 5.01s. This looks quite good for sendmail 8, but the result for:

`time ./smtp-source -c -s 100 -m 200 localhost:1234`

is: 143.12s, which certainly is not anywhere near good. This is related to the high load generated by this: up to 200 concurrent sendmail processes just kill the machine. sendmail X has only up to 4 processes running.

5.1.2.1 Various Linux FS

Test date: 2003-05-25, version: smX.0.0.6, machine: PC, AMD Duron 700MHz, 512MB RAM, SuSE 8.1

Test program:

```
time ./smtp-source -s 50 -m 500 -fa@b.c -tx@y.z localhost:1234
```

FS	Times	msg/s (best)
JFS	4.02s, 4.23s	124
ReiserFS	4.8s	104
XFS	6.7s, 7.2s, 7.48s, 7.64s	74
EXT3	14.39s, 13.44s	34

2004-03-17 `checks/t-readwrite` on destiny (Linux, IDE, ext2):

parameters	writes	time
-s -f 1000 -p 1	-	9
-s -f 100 -p 10	-	6

The FS is mounted async (default!).

2004-03-17 `checks/t-readwrite` on ia64-2 (Linux, SCSI, reiserfs):

parameters	writes	time
-s -f 1000 -p 1	-	5.2
-s -f 100 -p 10	-	2.6

2004-03-23 source: basil.ps-lab MTA: cilantro.ps-lab (Linux 2.4.18-64GB-SMP) sink: v-sun.perf-lab

FS: ReiserFS version 3.6.25

smtpc -t 100 -s 1000

program	source time	sink time
smtps3 -C		-
smX.0.0.12	6	5
sm8.12.11	74	74
sm8.12.11 See 1		50
postfix 2.0.18		

gatling -m 100 -c 5000 -z 1 -Z 1

program	writes	source time	source msgs/s	sink time
smtps3		2	2295	-
smtps3 -C		5	962	-
smX.0.0.12		22	225	22
sm8.12.11		358	14	358
sm8.12.11 See 1		246	20	-
postfix 2.0.18				

Notes:

1. Default for Linux is to have `REQUIRES_DIR_FSYNC` set, in this test it has been turned off. Some people claim it is safe to do that with recent Linux FSs. For some reasons (timeouts?) the tests with smtpc fail in this configuration, i.e., less than 1000 messages are sent.

2. According to tests by Thom sendmail 8.12 was able to relay 40 msgs/s on the same machine.

2004-03-25:

Filesystems:

1. ext3 (rw,sync,data=journal)
2. ext3 (rw,data=journal) [this means async?]
3. reiserfs (rw,noatime,data=journal,notail)
4. jfs (rw)
5. ext2 (rw,sync)

smtpc -t 100 -s 1000

program	FS	source time	sink time
smX.0.0.12	1	63	61
	1	63	63
	2	19	18
	3	5	4
	3	5	5
	5	81	80
sm8.12.11	3	45	several read errors
	5	91	92
smtps3 -C			

2004-03-25: gatling -m 100 -c 5000 -z 1 -Z 1 (1KB message size)

program	FS	source time	sink time	msgs/s
smX.0.0.12	1			
	2	90	90	55
	3	24	24	208
	4	100	99	100
sm8.12.11	3	216	errors	23

gatling -m 100 -c 5000 -z 4 -Z 4 (4KB message size)

program	FS	source time	sink time	msgs/s
smX.0.0.12	1			
	2	92	92	54
	3	141	140	35
	4	168	168	29
sm8.12.11	3	226	errors	22

gatling -m 100 -c 5000 -z 16 -Z 16 (16KB message size)

program	FS	source time	sink time	msgs/s
smX.0.0.12	1			
	2			
	3	169		29
	4			
sm8.12.11	3	226	errors	22

Notes:

1. ReiserFS seems to have some optimizations for small files, hence the results for 1KB are really good, but for 4KB they are in the *normal* range.
2. Testing with sm8 usually caused several read errors on the sink side and several errors displayed by gatling.

5.1.2.2 Various FreeBSD Results

2003-11-19 sm-9.0.0.9 running on v-bsd.perf-lab (2 processors, FreeBSD 3.4)

Source on bsd.dev-lab

time ./smtp-source -d -s 100 -m 500

directly to sink: 2.16 - 2.74s (231msgs/s)

using MFS: 14.37 - 14.43s (34msgs/s) (sm8.12.10: 32s)

using FS with softupdates: 22.78 - 23.83s (21msgs/s) (sm8.12.10: 49s)

using FS without softupdates: 35.27 - 35.56s (14msgs/s)

2004-03-02 source: s-6.perf-lab; relay: mon; sink: v-bsd

time ./smtpc2 -O 10 -fa@s-6.perf-lab -Rnobody@v-bsd.perf-lab -t 100 -s 1000 -r mon.perf-lab:1234

38.26s real 1.01s user 0.88s system

2004-03-04 source: s-6.perf-lab; relay: v-bsd; sink: v-sun

options: -t 100 -s 1000

MTA	source time(s)	sink time
postfix 2.0.18	53	94
smX.0.0.12	69	68
without smtpc	56	-
sm8.12.11	67	67
-odq	79, 82	
-odq / 100 qd	101	
-odq / 10 qd	100	

Note: this is FreeBSD 3.4 without softupdates and directory hashes.

`getrusage(2)` data:

sm8.12.11 -odq

```
ru_utime=        15.0158488
ru_stime=        71.0104605
ru_maxrss=     1524
ru_ixrss=   5030592
ru_idrss=   4098456
ru_isrss=   1412096
ru_minflt=   127503
ru_majflt=        0
```

```
ru_nswap=         0
ru_inblock=       0
ru_oublock=   11851
ru_msgsnd=    13000
ru_msgrcv=    10000
ru_nsignals=      0
ru_nvcsw=    617469
ru_nivcsw=    18793
```

sm8.12.11

```
ru_utime=        15.0236311
ru_stime=        62.0117941
ru_maxrss=     1520
ru_ixrss=   4573224
ru_idrss=   3676784
ru_isrss=   1283712
ru_minflt=   174619
ru_majflt=        0
ru_nswap=         0
ru_inblock=       0
ru_oublock=    4001
ru_msgsnd=    12000
ru_msgrcv=    10000
ru_nsignals=   1000
ru_nvcsw=    128074
ru_nivcsw=    14771
```

This looks like a problem in queue only mode: there's way too much data written: almost 3 times the amount of background delivery mode. Why does sm8 send 1000 more message in queue only mode?

2004-03-05 source, relay, sink: wiz (FreeBSD 4.8)

options: -t 100 -s 1000

source: 34s, sink: 32s

turn off smtpc: source: 31s, 34s

2004-03-26 source: v-6.perf-lab running `smtpc -t 100 -s 5000`; relay: v-bsd.perf-lab; sink: v-sun.perf-lab

sink runs `smtps2 -R` n with varying values for n

n	source time	requests served
0	108	5000
8000000	115	5060
58000000	140	5450
88000000	151	5620

put `defedb` on a RAM disk:

n	source time	requests served
0	108	5000
8000000		
58000000	111	5453
88000000	114	5693

Obviously the additional disk I/O traffic created by having to use DEFEDB is slowing down the system.

5.1.2.2.1 FreeBSD 4.9, Softupdates, and fsync() 2004-06-23 Upgraded v-bsd.perf-lab to FreeBSD 4.9 (2 processors), using softupdates.

source on v-sun, sink on s-6:

time ./smtpc2 -O 10 -t 100 -s 1000 -r v-bsd.perf-lab:1234

43s

turn off fsync(): (smtps -U, must be compiled with -DTESTING)

32s

5.1.2.2.2 Disk I/O On FreeBSD A modified `iostat(8)` program is used to show the number of bytes written and read, and the number of read, write, and other disk I/O operations.

The following tests were performed: sink (smtps3) on v-bsd.perf-lab, source (smtpc) on s-6.perf-lab sending 1000 mails. All numbers for write operations are rounded; if there are numbers in parentheses then those denote the value of `ru_oublock` (`getrusage(2)`) for smtps/qmgr or sm8. If two times are given (separated by /) then the second time denotes the output (elapsed time) for the sink.

program	softupdates?	writes	reads	time
smtps3 -C	yes	2200	-	14
smtps3 -C	no	2900	-	30
smX.0.0.12, no sched (see 1)	yes	5200	-	34
smX.0.0.12, no sched	yes		-	
smX.0.0.12, no sched	no		-	
smX.0.0.12 (see 2)	yes	3500 (2000/1300)	4	33
	yes	3370 (2020/1270)	4	30/29
-O i=1000000	yes	2660 (1850/660)	0	25/24
smX.0.0.12	no	6300 (3000/3200)	0	52
smX.0.0.12 (see 4)	yes	3500 (2200/1200)	4	25
sm8.12.11 -odq SS=m	yes	1800	-	41
sm8.12.11 -odq SS=m	no	12200	-	72
sm8.12.11 SS=m (see 3)	yes	236 (164)	0	61
	yes	370 (218)	0	60
sm8.12.11	no	8100 (4100)	1	63
sm8.12.11 SS=t	yes	7400	0	70
postfix 2.0.18	yes	2900	16	21/26

Notes:

1. *Question*: why does the smX.0.0.12 use so many write operations? 5200 is way too much. Answer:

qmgr committed IBDB more than 1000 times[1], increasing the maximum time to acknowledge an SMTPS transaction from 100 μs to 10000 μs reduces the number of commits to 165.

2. *Question*: why does qmgr still (after increasing the time between commits) perform so many write operations?

3. *Question*: why does sm8 use so few writes? Can softupdates eliminate or cluster most writes? Why doesn't this work for smX? Solution: `SuperSafe` was set to `m`, not to `true`.

4. If IBDB and CDB are on different partitions, the performance increases significantly (about 25 per cent faster).

2004-03-23 source: basil.ps-lab MTA: wasabi.ps-lab (FreeBSD 4.9, machine 16 in Section 5.2.1.1) sink: v-sun.perf-lab

smtpc -t 100 -s 1000

program	writes	reads	source time	sink time
smtps3 -C	2400	-	11	-
smX.0.0.12	2600	5	15	13
sm8.12.11	6000	1	35	
postfix 2.0.18	2800	15	14	20

Note: the sink time for postfix is shorter than the time for smX because smX emptied the queue during the run while postfix has more than 700 entries in the mail queue after the source finished sending all mails. This can be seen by looking at the *sink time* which is noticeable larger for postfix compared to sendmail X.

Using gatling:

```
Max random envelope rcpts:  1
Connections:                100
Max msgs/conn:              Unlimited
Messages:                   Fixed size 1 Kbytes
Desired Message Rate:       Unlimited
Total messages:             5000

Total test elapsed time: 73.571 seconds (1:13.570)
Overall message rate: 67.962 msg/sec
Peak rate: 100.000 msg/sec
```

gatling -m 100 -c 5000 -z 1 -Z 1

program	writes	source time	source msgs/s	sink time
smtps3	0	5	980	-
smtps3 -C	11750	53	93	
smX.0.0.12		73	67	71
smX.0.0.12	11157 (8000/2700)	70	71	69
sm8.12.11		136	36	
postfix 2.0.18		60	83	78
postfix 2.0.18	12635	58	85	75

[1] How can it be more than the number of transactions?

2004-03-16 results for wiz: source: `time ./smtpc -s 1000 -t 100 -r localhost:1234`; sink: smtps3, file system: UFS, softupdates

parameters	oublock	writes	source time	sink time
-C -i	1920	?	17	16
-C -p 1	1860	?	17	17
-C -p 1	1940	2700	16	15
-C -p 1	1970	2770	16	15
-C -p 2		?	15	?
-C -p 2	877+966	2600	15	?
-C -p 4	455+476+432+472	2640	15	?

New option: `-f` for *flat*, i.e., instead of using 16 subdirectories for CDB files, a single directory is used. Even though this does not cause a noticeable difference in run time, the number of I/O operations is reduced.

parameters	oublock	writes	source time
-C -p 2	915+920	2600	14
-C -p 2 -f	600+610	2200	14

2004-03-16 source: s-6.perf-lab, `time ./smtpc -s 1000 -t 100 -r localhost:1234`; sink: -v-bsd.perf-lab, smtps3, file system: UFS, softupdates

parameters	oublock	writes	source time	sink time
-C -i	1430	2165	12	11
	1550	2300	14	13
-C -p 1	1500	2500	14	12
-C -p 2	1100+620	2500	13	-
	800+770	2320	13	-
-C -p 4	530+350+540+470	2600	13	-

Note: some of the write operations might be from softupdates due to the previous `rm` command (removing the CDB files).

2004-03-17 `checks/t-readwrite` on v-bsd (FreeBSD 4.9, SCSI):

parameters	softupdates	oublock	writes	time
-s -f 1000 -p 1	yes	4000	4000	22
-s -f 100 -p 10	yes	2575	2579	14
-s -f 1000 -p 1	no	4050	4050	28
-s -f 100 -p 10	no	4050	4050	27

`-p` specifies the number of processes to start, `-f` specifies the number of files to write per process. The test cases above write 1000 files with either 1 or 10 processes. As it can be seen, it is significantly more efficient to use 10 processes if softupdates are turned on.

2004-03-17 `checks/t-readwrite` on wiz (FreeBSD 4.8, IDE):

parameters	softupdates	oublock	writes	time
-s -f 1000 -p 1	yes	3000	3800	13
-s -f 100 -p 10	yes	2860	3600	13

In this case no difference can be seen, which is most likely a result of using an IDE drive with write-caching turned on (default).

5.1.2.3 Various SunOS 5 Results

2003-11-21 sm-9.0.0.9 running on v-sun.perf-lab

Source on bsd.dev-lab

time ./smtp-source -d -s 100 -m 5000 -c

using FS: 301.90 - 305.02s (16msgs/s)

using swap: 77.98 - 78.55s (64msgs/s)

Those tests ran only 32 SMTPS threads (the machine has 4 CPUs, hence the specified limit 128 was divided by 4). Using 128 SMTPS threads (by forcing only one process which was used anyway because SMTPS is run with the interactive option which does not start backgroup processes):

time ./smtp-source -d -s 100 -m 50000 -c

using swap: 727.73s (68msgs/s)

2004-03-09 sm-9.0.0.12 running on v-sun.perf-lab

time ./smtpc -O 20 -fa@s-6.perf-lab.sendmail.com -Rnobody@v-bsd.perf-lab.sendmail.com -t 100 -s 1000 -r v-sun.perf-lab.sendmail.com:1234

MTA options	FS	source time(s)	sink time(s)
full MTS	SWAPFS	16	14
without sched	SWAPFS	10	-
smtpss	SWAPFS	3	-
full MTS	UFS	64, 65, 64	75, 70, 69
8.12.11	SWAPFS	16	19
8.12.11	UFS	141	138

Note: smX using UFS runs into connection limitations: QMGR *believes* there are 100 open connections even though the sink shows at most 18. This seems to be a communication latency between SMTPC and QMGR (and needs to be investigated further).

2004-03-17 `checks/t-readwrite` on v-sun (SunOS 5.8, SCSI):

parameters	writes	time
-s -f 1000 -p 1	-	39
-s -f 100 -p 10	-	37

The filesystem on SunOS 5.8 does not cause any difference whether 1 or 10 processes are used.

5.1.2.4 Various OpenBSD Results

2004-03-05 source, relay, and sink on zardoc (OpenBSD 3.2)

test with logging via smioout

```
zardoc$ time ./smtpc2 -O 10 -s 1000 -t 100 -r localhost:1234
   24.17s real     0.94s user     2.57s system
```

smtps3 stats:

```
elapsed                    26
```

```
Thread limits (min/max)    8/256
Waiting threads            8
Max busy threads           3
Requests served            1000
```

Note that there have been only 3 active threads. That means the client is not busy at all. Another test shows elapsed=23s, max busy threads=21, so the result isn't deterministic (the machine is running as normal SMTP server etc during tests).

test with logging via smioerr: smtpc2: 24.53s; no difference.

5.1.2.5 Various AIX Results

2004-03-17 `checks/t-readwrite` on aix-3 (AIX 4.3, SCSI, jfs):

parameters	writes	time
-s -f 1000 -p 1	-	30
-s -f 100 -p 10	-	29

No (noticeable) difference.

5.2 Implementation of Queues and Caches

5.2.1 Filesystem Performance

Here are some results of a simple test program which creates and deletes a number of files and optionally renames them twice while doing so.

Notice: unless mentioned otherwise, all measurements are at most accurate to one second resolution. Repeated test will most likely show (slightly) different results. These tests are only listed to give an idea of the magnitude of available performance.

5.2.1.1 Test Systems

The involved systems are:

1. PC, Pentium III, 500MHz, 256MB RAM, FreeBSD 3.2,

   ```
   wdc0: unit 0 (wd0): <FUJITSU MPD3064AT>
   wd0: 6187MB (12672450 sectors), 13410 cyls, 15 heads, 63 S/T, 512 B/S
   ```

2. PC, AMD K6-2, 450MHz, 220MB RAM, OpenBSD 2.8,

   (a)
   ```
   wd0 at pciide0 channel 0 drive 0: <IBM-DJNA-351010>
   wd0: can use 32-bit, PIO mode 4, DMA mode 2, Ultra-DMA mode 4
   wd0: 16-sector PIO, LBA, 9671MB, 16383 cyl, 16 head, 63 sec, 19807200 sectors
   ```

   (b)
   ```
   wd1 at pciide0 channel 0 drive 1: <Maxtor 98196H8>,
   wd1: can use 32-bit, PIO mode 4, DMA mode 2, Ultra-DMA mode 4,
   wd1: 16-sector PIO, LBA, 78167MB, 16383 cyl, 16 head, 63 sec, 160086528 sectors
   ```

3. PC, Pentium III, 500MHz, 256MB RAM, FreeBSD 4.4-STABLE,

   ```
   ad0: 6187MB <FUJITSU MPC3064AT> [13410/15/63] at ata0-master UDMA33
   ```

4. PC, AMD-K7, 500MHz, FreeBSD 4.4-STABLE, 332MB RAM,

   (a)
   ```
   ahc0: <Adaptec 2940 Ultra2 SCSI adapter (OEM)>
   da0: <IBM DNES-309170W SA30> Fixed Direct Access SCSI-3 device
   da0: 40.000MB/s transfers (20.000MHz, offset 31, 16bit), Tagged Queueing Enabled
   da0: 8748MB (17916240 512 byte sectors: 255H 63S/T 1115C)
   ```
      i. SCSI with softupdates
      ii. SCSI without softupdates

   (b)
   ```
   ad0: 8063MB <FUJITSU MPD3084AT> [16383/16/63] at ata0-master UDMA66
   ```
   softupdates

5. PC, Linux 2.2.12,

   ```
   hda: IBM-DJNA-370910, 8693MB w/1966kB Cache, CHS=1108/255/63
   ```

   ext 2 FS

6. PC, Linux 2.4.7,

   ```
   hda: 39102336 sectors (20020 MB) w/2048KiB Cache, CHS=2434/255/63, UDMA(66)
   reiserfs: using 3.5.x disk format
   ReiserFS version 3.6.25
   ```

7. Dec Digitial Alpha, OSF/1, SCSI disk?

8. Sun SPARC, SunOS 5.6, SCSI disk?

9. Sun SPARC, SunOS 5.7, SCSI disk?
   (a) mount options: no logging, atime
   (b) mount options: logging, atime
   (c) mount options: logging, noatime

10. Sun SPARC E450, 4 CPUs,
    (a) Baydel RAID
    (b) SCSI disk

11. AIX 4.3.3, using JFS (default).

12. PC, AMD K7, 1000MHz, 512MB RAM, SuSE 7.3, kernel 2.4.10

    ```
    WD1200BB
    hdg: 234441648 sectors (120034 MB) w/2048KiB Cache, CHS=232581/16/63, UDMA(100)
    ```

    (a) /home jfs
    (b) /opt reiserfs
    (c) /work ext3

13. HP-UX 11.00

14. PC, Pentium II, 360MHz, 512MB RAM, FreeBSD 4.6,

```
ad0: 8693MB <IBM-DJNA-370910> [17662/16/63] at ata0-master UDMA33
acd0: CDROM <CD-ROM 40X> at ata1-master PIO4
```

15. Intel IA64, 4 CPUs, 1GB RAM

```
scsi0 : ioc0: LSI53C1030, FwRev=01000000h, Ports=1, MaxQ=255, IRQ=52
  Vendor: MAXTOR    Model: ATLASU320_18_SCA  Rev: B120
  Type:   Direct-Access                      ANSI SCSI revision: 03
Attached scsi disk sda at scsi0, channel 0, id 0, lun 0
SCSI device sda: 35916548 512-byte hdwr sectors (18389 MB)
reiserfs: found format "3.6" with standard journal
reiserfs: using ordered data mode
Using r5 hash to sort names
```

16. Intel Pentium III, 650 MHz, 256MB RAM

```
da0 at ahc0 bus 0 target 0 lun 0
da0: <SEAGATE ST39175LW 0001> Fixed Direct Access SCSI-2 device
da0: 80.000MB/s transfers (40.000MHz, offset 15, 16bit), Tagged Queueing Enabled
da0: 8683MB (17783240 512 byte sectors: 255H 63S/T 1106C)
```

17. PC, Pentium III, 450MHz, 256MB RAM, FreeBSD 4.8, softupdates

```
ad0: 6187MB <FUJITSU MPD3064AT> [13410/15/63] at ata0-master UDMA33
```

18. PC, FreeBSD 4.10, softupdates

```
da3 at ahc0 bus 0 target 4 lun 0
da3: <IBM DNES-309170Y SA30> Fixed Direct Access SCSI-3 device
da3: 40.000MB/s transfers (20.000MHz, offset 31, 16bit), Tagged Queueing Enabled
da3: 8748MB (17916240 512 byte sectors: 255H 63S/T 1115C)
```

19. PC, VIA C3, 667MHz, 256MB RAM, OpenBSD 3.2,

   (a) `wd0 at pciide0 channel 0 drive 0: <IBM-DJNA-371350>`
   `wd0: 16-sector PIO, LBA, 12949MB, 16383 cyl, 16 head, 63 sec, 26520480 sectors`

   (b) `wd1 at pciide0 channel 0 drive 1: <WDC WD1200BB-53CAA0>`
   `wd1: 16-sector PIO, LBA, 114473MB, 16383 cyl, 16 head, 63 sec, 234441648 sectors`

   (c) `wd2 at pciide1 channel 0 drive 0: <Maxtor 6Y160P0>`
   `wd2: 16-sector PIO, LBA48, 156334MB, 16383 cyl, 16 head, 63 sec, 320173056 sectors`
   `wd2(pciide1:0:0): using PIO mode 4, Ultra-DMA mode 6`

5.2.1.2 Meta Data Operations

In this section, some simple test programs are used that create some files, perform (sequential) read/write operations on them and remove them afterwards.

Entries in the following table are elapsed time in seconds (except for the first column which obviously refers to the machine description above). The program that has been used to produce these results is `fsperf1.c`.

machine	5000 100	-c 5000 100	-c -r 5000 100
1	50	49	48
1	42	48	51
2a	3	7 about 2200 tps	10 about 1500 tps
2b		11	21
3	10 about 500 tps	34	34
4(a)i		126	125
4(a)ii		208	454
4b		43	48
7	7	13	16
5	9	8	9
8	133	201	603
9a		52	665
10a	9	9	12
11	89	139	233

Comments:

- 2a is probably so fast due to a disk cache, i.e., the data is probably not really written to disk (even though fsync() is used). The same might hold for 5. In case of 10a the RAID system has a (battery backed) RAM disk cache, which gives similar results without the risk of losing data in case of a power loss.
- 8 is unacceptably slow...

(2004-07-14) With and without `fsync(2)` (-S)

common parameters	machine	-c	-c -r	-S -c	-S -c -r
(5000 100)	17	42	42	2	3
	10b	165	496	165	495
	18	83	83	5	8
	19a	8	7	1	3
	19b	8	9	1	3
	19c	7	9	1	2
(-s 32 5000 100)	17	109	109	8	9
	10b	250	537	207	498
	18	114	113	14	16
	19b	87	81	3	5
	19c	26	26	4	5

Comments:

- Sun's FS is unbelievable slow. Moreover, it doesn't make a difference whether `fsync(2)` is used or not. That means the FS does not have any optimization like softupdates or journalling FSs have.
- FreeBSD softupdate can optimize renaming in this test.
- IDE is faster than SCSI for small sizes because of the cache; it most likely "cheats" (write cache is not disabled, hence the date may not really be on disk even if `fsync(2)` is used).

Next version: allow for hashing (00 - 99, up to two levels). Use enough files to defeat the (2MB) cache of IDE disks.

machine	-h 1 -c 1000 1000	-h 1 -c -r 1000 1000
1	18	18
2a	24	24
2b	7	9
3	14	14
4(a)i	23	23
4(a)ii	33	77
4b	25	49
5	3	2
7	3	4
8	58	163
9a	51	139
11	28	48

Comments:

- The fact that there is no difference for the two tests for 2a might be again be attributed to the disk cache. The same effect can be observed for 1, 3, and 5.
- The result for 4(a)i (softupdates) indicates that softupdates eliminates the two consecutive rename(2)s since no fsync(2) is issued inbetween. The results without softupdates 4(a)ii reflect this.
- 8, 9a, and 11 show the normal filesystem (UFS) behaviour.
- 7 is extremely fast.

5.2.1.2.1 Meta Data Operations: Existing Files Next version fsperf1.c: allow for hashing (00 - 99, up to two levels). Use enough files to defeat the (2MB) cache of IDE disks. The parameters for the following table are 1000 operations and 1000 files, hence each file is used once. Additional parameters are listed in the heading. c: create, h 1: one level hashing, r: rename file twice, p: populate directories before test, then just reuse the files.

machine	-h 1 -c	-h 1 -c -r	-p -h 1 -c	-p -h 1 -c -r
1	32	31	18	17
2a	18	18	9	10
2b	10	10	8	10
5	2	1	2	1
6	2	2	4	4
7	2	4	2	3
8	58	165	78	178
9a	27	127	33	131
9c	13	51	37	55
11	28	48	28	48

Comments:

- 5 must be cheating (ext2, async).
- Why are populated directories slower on 8?
- Using logging (and noatime) on 9 makes rename() more than two times faster.

5.2.1.3 Writing a Logfile

Another test program (fsseq1.c) writes lines to a file and uses fsync(2) after a specified number (-C parameter).

20000 entries (10000 entries each for received/delivered, total 490000 bytes).

machine	-	-C 100	-C 50	-C 10	-C 5	-C 2	-f
1	1	4	6	17	32	78	150
2a	0	2	2	5	5	9	18
2b	1	0	1	3	4	10	20
3	1	2	3	9	16	37	68
5	1	1	2	6	12	27	56
7	0	4	8	39	79	198	410
8	1	7	13	60	120	299	598
9a	1	8	13	15	62	90	140
11	0	6	12	53	106	262	518

This clearly demonstrates the need for group commits. However, the program requires a lot of CPU since each line is generated by snprintf(). Hence the full I/O speed may not be reached. To confirm this, another program (fsseq2.c) is used that just writes a buffer with a fixed content to a file.

The following table lists the results for group commits (C) together with various buffer sizes (256, 1024, 4096, 8192, and 16384). As usual the entries are execution time in seconds. The program writes 2000 records in total, e.g., for size 16384 that is 31MB data.

machine	C	256	1024	4096	8192	16384
5	1	4	5	10	20	34
	2	2	4	6	12	22
	5	1	2	5	7	15
	10	1	1	3	6	12
	50	1	0	3	5	10
	100	0	1	3	5	10
7	1	1	5	20	40	44
	2	1	5	11	23	29
	5	1	5	9	12	13
	10	1	2	3	6	7
	50	0	1	1	2	3
	100	0	1	1	1	3
8	1	3	10	45	95	109
	2	2	11	23	52	59
	5	3	11	19	24	32
	10	2	5	6	15	21
	50	1	2	3	8	13
	100	0	1	3	6	13
9a	1	3	12	34	35	58
	2	3	12	18	53	53
	5	3	6	21	23	24
	10	3	5	6	13	14
	50	1	2	2	5	7
	100	1	1	2	3	6
11	1	21	35	77	83	92
	2	13	26	38	45	50
	5	8	13	17	20	24
	10	5	6	10	11	15
	50	1	2	2	4	7
	100	1	1	2	3	6

Comments:

- 7 is able to sustain about 10MB/s write rate, 9a reaches about 5MB/s, just like 11.

- 5 achieves about 3MB/s.

Yet another program (fsseq3.c) uses write() instead of fwrite(). This time the tests write 40000KB each, which makes it simpler to determine the throughput.

Note: as usual, these times are not very accurate (1s resolution), and hence the rate is inaccurate too. Machines:

1

C	s	records	time	KB/s
1	512	80000	1365	29
1	1024	40000	734	54
1	2048	20000	451	88
1	4096	10000	352	113
1	8192	5000	250	160
2	512	80000	736	54
2	1024	40000	453	88
2	2048	20000	354	112
2	4096	10000	382	104
2	8192	5000	225	177
5	512	80000	638	62
5	1024	40000	585	68
5	2048	20000	312	128
5	4096	10000	187	213
5	8192	5000	101	396
10	512	80000	561	71
10	1024	40000	296	135
10	2048	20000	161	248
10	4096	10000	88	454
10	8192	5000	60	666
50	512	80000	128	312
50	1024	40000	70	571
50	2048	20000	41	975
50	4096	10000	34	1176
50	8192	5000	29	1379
100	512	80000	73	547
100	1024	40000	43	930
100	2048	20000	33	1212
100	4096	10000	28	1428
100	8192	5000	27	1481

2b

C	s	records	time	KB/s
1	512	80000	165	242
1	1024	40000	90	444
1	2048	20000	54	740
1	4096	10000	28	1428
1	8192	5000	16	2500
2	512	80000	94	425
2	1024	40000	52	769
2	2048	20000	30	1333
2	4096	10000	17	2352
2	8192	5000	11	3636
5	512	80000	54	740
5	1024	40000	33	1212
5	2048	20000	19	2105
5	4096	10000	11	3636
5	8192	5000	8	5000
10	512	80000	31	1290
10	1024	40000	18	2222
10	2048	20000	11	3636
10	4096	10000	8	5000
10	8192	5000	6	6666
50	512	80000	11	3636
50	1024	40000	8	5000
50	2048	20000	6	6666
50	4096	10000	5	8000
50	8192	5000	4	10000
100	512	80000	10	4000
100	1024	40000	8	5000
100	2048	20000	5	8000
100	4096	10000	4	10000
100	8192	5000	5	8000

5

C	s	records	time	KB/s
1	512	80000	13440	2
1	1024	40000	6790	5
1	2048	20000	3451	11
1	4096	10000	1779	22
1	8192	5000	1007	39
2	512	80000	6790	5
2	1024	40000	3439	11
2	2048	20000	1763	22
2	4096	10000	909	44
2	8192	5000	471	84
5	512	80000	2763	14
5	1024	40000	1414	28
5	2048	20000	739	54
5	4096	10000	383	104
5	8192	5000	208	192
10	512	80000	1414	28
10	1024	40000	731	54
10	2048	20000	384	104
10	4096	10000	208	192
10	8192	5000	120	333
50	512	80000	312	128
50	1024	40000	174	229
50	2048	20000	101	396
50	4096	10000	64	625
50	8192	5000	46	869
100	512	80000	171	233
100	1024	40000	100	400
100	2048	20000	64	625
100	4096	10000	46	869
100	8192	5000	37	1081

6

C	s	records	time	KB/s
1	512	80000	130	307
1	1024	40000	93	430
1	2048	20000	78	512
1	4096	10000	23	1739
1	8192	5000	12	3333
2	512	80000	62	645
2	1024	40000	46	869
2	2048	20000	24	1666
2	4096	10000	13	3076
2	8192	5000	15	2666
5	512	80000	66	606
5	1024	40000	31	1290
5	2048	20000	18	2222
5	4096	10000	15	2666
5	8192	5000	10	4000
10	512	80000	28	1428
10	1024	40000	19	2105
10	2048	20000	13	3076
10	4096	10000	10	4000
10	8192	5000	10	4000
50	512	80000	14	2857
50	1024	40000	10	4000
50	2048	20000	10	4000
50	4096	10000	9	4444
50	8192	5000	7	5714
100	512	80000	11	3636
100	1024	40000	10	4000
100	2048	20000	8	5000
100	4096	10000	8	5000
100	8192	5000	8	5000

7

C	s	records	time	KB/s
1	512	80000	3347	11
1	1024	40000	1689	23
1	2048	20000	845	47
1	4096	10000	418	95
1	8192	5000	192	208
2	512	80000	1243	32
2	1024	40000	796	50
2	2048	20000	431	92
2	4096	10000	222	180
2	8192	5000	122	327
5	512	80000	655	61
5	1024	40000	268	149
5	2048	20000	161	248
5	4096	10000	108	370
5	8192	5000	58	689
10	512	80000	355	112
10	1024	40000	185	216
10	2048	20000	85	470
10	4096	10000	42	952
10	8192	5000	38	1052
50	512	80000	88	454
50	1024	40000	49	816
50	2048	20000	31	1290
50	4096	10000	18	2222
50	8192	5000	10	4000
100	512	80000	45	888
100	1024	40000	33	1212
100	2048	20000	19	2105
100	4096	10000	14	2857
100	8192	5000	14	2857

8

C	s	records	time	KB/s
1	512	80000	6302	6
1	1024	40000	3220	12
1	2048	20000	1695	23
1	4096	10000	949	42
1	8192	5000	552	72
2	512	80000	3183	12
2	1024	40000	1708	23
2	2048	20000	950	42
2	4096	10000	484	82
2	8192	5000	299	133
5	512	80000	1402	28
5	1024	40000	805	49
5	2048	20000	440	90
5	4096	10000	252	158
5	8192	5000	137	291
10	512	80000	783	51
10	1024	40000	395	101
10	2048	20000	211	189
10	4096	10000	122	327
10	8192	5000	87	459
50	512	80000	181	220
50	1024	40000	107	373
50	2048	20000	68	588
50	4096	10000	49	816
50	8192	5000	42	952
100	512	80000	111	360
100	1024	40000	70	571
100	2048	20000	50	800
100	4096	10000	40	1000
100	8192	5000	36	1111

9a

C	s	records	time	KB/s
1	512	80000	2638	15
1	1024	40000	1419	28
1	2048	20000	753	53
1	4096	10000	442	90
1	8192	5000	221	180
2	512	80000	1379	29
2	1024	40000	774	51
2	2048	20000	409	97
2	4096	10000	220	181
2	8192	5000	124	322
5	512	80000	644	62
5	1024	40000	382	104
5	2048	20000	198	202
5	4096	10000	105	380
5	8192	5000	58	689
10	512	80000	355	112
10	1024	40000	196	204
10	2048	20000	104	384
10	4096	10000	59	677
10	8192	5000	32	1250
50	512	80000	90	444
50	1024	40000	51	784
50	2048	20000	28	1428
50	4096	10000	19	2105
50	8192	5000	15	2666
100	512	80000	54	740
100	1024	40000	28	1428
100	2048	20000	20	2000
100	4096	10000	15	2666
100	8192	5000	14	2857

9b

C	s	records	time	KB/s
1	512	80000	2642	15
1	1024	40000	1312	30
1	2048	20000	723	55
1	4096	10000	376	106
1	8192	5000	185	216
2	512	80000	1363	29
2	1024	40000	699	57
2	2048	20000	359	111
2	4096	10000	185	216
2	8192	5000	104	384
5	512	80000	563	71
5	1024	40000	302	132
5	2048	20000	162	246
5	4096	10000	88	454
5	8192	5000	46	869
10	512	80000	299	133
10	1024	40000	161	248
10	2048	20000	87	459
10	4096	10000	46	869
10	8192	5000	24	1666
50	512	80000	81	493
50	1024	40000	44	909
50	2048	20000	35	1142
50	4096	10000	19	2105
50	8192	5000	13	3076
100	512	80000	51	784
100	1024	40000	35	1142
100	2048	20000	26	1538
100	4096	10000	15	2666
100	8192	5000	13	3076

9c

C	s	records	time	KB/s
1	512	80000	2576	15
1	1024	40000	1326	30
1	2048	20000	707	56
1	4096	10000	377	106
1	8192	5000	192	208
2	512	80000	1324	30
2	1024	40000	685	58
2	2048	20000	349	114
2	4096	10000	187	213
2	8192	5000	107	373
5	512	80000	578	69
5	1024	40000	313	127
5	2048	20000	163	245
5	4096	10000	89	449
5	8192	5000	46	869
10	512	80000	306	130
10	1024	40000	162	246
10	2048	20000	86	465
10	4096	10000	46	869
10	8192	5000	25	1600
50	512	80000	82	487
50	1024	40000	44	909
50	2048	20000	33	1212
50	4096	10000	19	2105
50	8192	5000	13	3076
100	512	80000	52	769
100	1024	40000	36	1111
100	2048	20000	25	1600
100	4096	10000	16	2500
100	8192	5000	13	3076

12a

C	s	records	time	KB/s
1	512	80000	65	615
1	1024	40000	61	655
1	2048	20000	59	677
1	4096	10000	5	8000
1	8192	5000	4	10000
2	512	80000	13	3076
2	1024	40000	8	5000
2	2048	20000	4	10000
2	4096	10000	4	10000
2	8192	5000	3	13333
5	512	80000	44	909
5	1024	40000	21	1904
5	2048	20000	13	3076
5	4096	10000	3	13333
5	8192	5000	3	13333
10	512	80000	12	3333
10	1024	40000	3	13333
10	2048	20000	3	13333
10	4096	10000	3	13333
10	8192	5000	5	8000
50	512	80000	11	3636
50	1024	40000	3	13333
50	2048	20000	5	8000
50	4096	10000	5	8000
50	8192	5000	4	10000
100	512	80000	5	8000
100	1024	40000	5	8000
100	2048	20000	5	8000
100	4096	10000	4	10000
100	8192	5000	3	13333

12b

C	s	records	time	KB/s
1	512	80000	124	322
1	1024	40000	87	459
1	2048	20000	72	555
1	4096	10000	20	2000
1	8192	5000	10	4000
2	512	80000	47	851
2	1024	40000	32	1250
2	2048	20000	16	2500
2	4096	10000	8	5000
2	8192	5000	5	8000
5	512	80000	56	714
5	1024	40000	27	1481
5	2048	20000	20	2000
5	4096	10000	5	8000
5	8192	5000	5	8000
10	512	80000	23	1739
10	1024	40000	17	2352
10	2048	20000	6	6666
10	4096	10000	3	13333
10	8192	5000	6	6666
50	512	80000	7	5714
50	1024	40000	4	10000
50	2048	20000	6	6666
50	4096	10000	6	6666
50	8192	5000	4	10000
100	512	80000	7	5714
100	1024	40000	6	6666
100	2048	20000	5	8000
100	4096	10000	4	10000
100	8192	5000	3	13333

12c

C	s	records	time	KB/s
1	512	80000	205	195
1	1024	40000	144	277
1	2048	20000	122	327
1	4096	10000	14	2857
1	8192	5000	7	5714
2	512	80000	34	1176
2	1024	40000	22	1818
2	2048	20000	13	3076
2	4096	10000	7	5714
2	8192	5000	5	8000
5	512	80000	96	416
5	1024	40000	48	833
5	2048	20000	20	2000
5	4096	10000	4	10000
5	8192	5000	4	10000
10	512	80000	36	1111
10	1024	40000	7	5714
10	2048	20000	5	8000
10	4096	10000	4	10000
10	8192	5000	3	13333
50	512	80000	12	3333
50	1024	40000	4	10000
50	2048	20000	4	10000
50	4096	10000	3	13333
50	8192	5000	3	13333
100	512	80000	7	5714
100	1024	40000	6	6666
100	2048	20000	3	13333
100	4096	10000	3	13333
100	8192	5000	3	13333

5.2.1.3.1 Raw Throughput Very simple measurement of transfer rate:

```
time dd ibs=8192 if=/dev/zero obs=8192 count=5120 of=incq
```

machine	s	MB/s
1	11.6	3.6
2a	4.8	8.4
2b	1.9	20.9
5	10.83	3.9
6	0.65	61
7	1.0	40.0
8	14.8	2.8
9	6.3	6.6
11	6.98	6.0
12a	0.247	161
12b	0.401	99
12c	0.357	112

Comments:

- The dd throughput is about twice as high as the maximum achieved via fsseq3 for 1, 2b, and 8.
- 5 seems to have a very bad fsync() implementation, and even in the best case the throughput is off by a factor of 3 from the dd value.
- For 7 the difference is about a factor of 13, which either means the dd time is flawed (pretty likely, since it is very short), or the fsync() calls are significant on that system.
- The dd throughput for 40MB is absurdely high on the Linux 2.4.x systems. That might be due to the default async mount option. See below for a test with a larger value

```
dd ibs=8192 if=/dev/zero obs=8192 count=124000 of=incq
```

machine	s	MB/s
12a	24.762	39
12b	22.608	42

The data in this table is more likely, even though 40MB/s is still very fast.

5.2.1.3.2 Writing a Logfile; 2nd Version For comparison with the Berkeley DB performance data, more tests have been run with fsseq4 with different parameters. Number of records is 100000 unless otherwise noted, t/s is transactions (records written) per second. Notice: fsseq3 writes twice as much records as fsseq4 (one add and one delete entry each), and it calls fsync() twice as often (after the add and after the delete entry).

	C	s	time	KB/s	t/s		C	s	time	KB/s	t/s
	100000	20	1	1953	100000		100000	20	1	1953	100000
	10000	20	2	976	50000		10000	20	1	1953	100000
	1000	20	7	279	14285		1000	20	2	976	50000
	100	20	20	97	5000		100	20	2	976	50000
	100000	100	3	3255	33333		100000	100	2	4882	50000
	10000	100	4	2441	25000		10000	100	1	9765	100000
	1000	100	8	1220	12500		1000	100	2	4882	50000
	100	100	57	171	1754		100	100	7	1395	14285
	100000	512	15	3333	6666		100000	512	3	16666	33333
1	10000	512	16	3125	6250	2b	10000	512	3	16666	33333
	1000	512	17	2941	5882		1000	512	4	12500	25000
	100	512	67	746	1492		100	512	6	8333	16666
	100000	1024	29	3448	3448		100000	1024	6	16666	16666
	10000	1024	30	3333	3333		10000	1024	5	20000	20000
	1000	1024	33	3030	3030		1000	1024	6	16666	16666
	100	1024	77	1298	1298		100	1024	8	12500	12500
	100000	2048	60	3333	1666		100000	2048	12	16666	8333
	10000	2048	60	3333	1666		10000	2048	12	16666	8333
	1000	2048	64	3125	1562		1000	2048	15	13333	6666
	100	2048	101	1980	990		100	2048	15	13333	6666

	C	s	time	KB/s	t/s
	100000	20	1	1953	100000
	10000	20	1	1953	100000
	1000	20	2	976	50000
	100	20	9	217	11111
	100000	100	3	3255	33333
	10000	100	4	2441	25000
	1000	100	5	1953	20000
	100	100	15	651	6666
	100000	512	16	3125	6250
5	10000	512	18	2777	5555
	1000	512	22	2272	4545
	100	512	75	666	1333
	100000	1024	34	2941	2941
	10000	1024	35	2857	2857
	1000	1024	46	2173	2173
	100	1024	139	719	719
	100000	2048	67	2985	1492
	10000	2048	79	2531	1265
	1000	2048	95	2105	1052
	100	2048	246	813	406

	C	s	time	KB/s	t/s
	100000	20	1	1953	100000
	10000	20	1	1953	100000
	1000	20	4	488	25000
	100	20	31	63	3225
	100000	100	2	4882	50000
	10000	100	2	4882	50000
	1000	100	6	1627	16666
	100	100	33	295	3030
	100000	512	8	6250	12500
7	10000	512	11	4545	9090
	1000	512	15	3333	6666
	100	512	50	1000	2000
	100000	1024	11	9090	9090
	10000	1024	10	10000	10000
	1000	1024	14	7142	7142
	100	1024	42	2380	2380
	100000	2048	25	8000	4000
	10000	2048	26	7692	3846
	1000	2048	21	9523	4761
	100	2048	42	4761	2380

	C	s	time	KB/s	t/s
	100000	20	3	651	33333
	10000	20	3	651	33333
	1000	20	3	651	33333
	100	20	5	390	20000
	100000	100	3	3255	33333
	10000	100	4	2441	25000
	1000	100	4	2441	25000
	100	100	9	1085	11111
	100000	512	5	10000	20000
8	10000	512	5	10000	20000
	1000	512	7	7142	14285
	100	512	20	2500	5000
	100000	1024	8	12500	12500
	10000	1024	8	12500	12500
	1000	1024	9	11111	11111
	100	1024	26	3846	3846
	100000	2048	15	13333	6666
	10000	2048	16	12500	6250
	1000	2048	21	9523	4761
	100	2048	36	5555	2777

	C	s	time	KB/s	t/s
	100000	20	1	1953	100000
	10000	20	1	1953	100000
	1000	20	4	488	25000
	100	20	29	67	3448
	100000	100	1	9765	100000
	10000	100	2	4882	50000
	1000	100	5	1953	20000
	100	100	36	271	2777
	100000	512	4	12500	25000
11	10000	512	5	10000	20000
	1000	512	9	5555	11111
	100	512	44	1136	2272
	100000	1024	8	12500	12500
	10000	1024	9	11111	11111
	1000	1024	13	7692	7692
	100	1024	54	1851	1851
	100000	2048	15	13333	6666
	10000	2048	17	11764	5882
	1000	2048	22	9090	4545
	100	2048	67	2985	1492

10a

C	s	time	KB/s	t/s
100000	20	2	976	50000
10000	20	1	1953	100000
1000	20	2	976	50000
100	20	3	651	33333
100000	100	2	4882	50000
10000	100	2	4882	50000
1000	100	2	4882	50000
100	100	6	1627	16666
100000	512	3	16666	33333
10000	512	3	16666	33333
1000	512	4	12500	25000
100	512	21	2380	4761
100000	1024	3	33333	33333
10000	1024	4	25000	25000
1000	1024	7	14285	14285
100	1024	41	2439	2439
100000	2048	4	50000	25000
10000	2048	5	40000	20000
1000	2048	12	16666	8333
100	2048	80	2500	1250

10b

C	s	time	KB/s	t/s
100000	20	1	1953	100000
10000	20	1	1953	100000
1000	20	4	488	25000
100	20	23	84	4347
100000	100	2	4882	50000
10000	100	2	4882	50000
1000	100	5	1953	20000
100	100	32	305	3125
100000	512	5	10000	20000
10000	512	5	10000	20000
1000	512	9	5555	11111
100	512	42	1190	2380
100000	1024	10	10000	10000
10000	1024	11	9090	9090
1000	1024	14	7142	7142
100	1024	59	1694	1694
100000	2048	21	9523	4761
10000	2048	21	9523	4761
1000	2048	25	8000	4000
100	2048	78	2564	1282

Comments:

- Using record sizes that are not a divider of the block size of the underlying filesystem reduces throughput since fsync() can't write whole blocks in that case.

- 10a behaves very bad for C=100. The reason for this is unknown. The times are especially bad compared to the Berkeley DB tests run on the same configuration (see Section 5.2.3).

5.2.2 Harddisk Performance

Some performance data gathered from the WWW.

SR Office DriveMark 2002 in IO/Sec taken from [Ra01]:

Manufacturer	Model	I/O operations/second
Seagate	Cheetah X15-36LP (36.7 GB Ultra160/m SCSI)	485
Maxtor	Atlas 10k III (73 GB Ultra160/m SCSI)	455
Fujitsu	MAM3367 (36 GB Ultra160/m SCSI)	446
IBM	Ultrastar 36Z15 (36.7 GB Ultra160/m SCSI)	402
Western Digital	Caviar WD1000BB-SE (100 GB ATA-100)	397
Seagate	Cheetah 36ES (36 GB Ultra160/m SCSI)	373
Fujitsu	MAN3735 (73 GB Ultra160/m SCSI)	369
Seagate	Cheetah 73LP (73.4 GB Ultra160/m SCSI)	364
Western Digital	Caviar WD1200BB (120 GB ATA-100)	337
Seagate	Cheetah 36XL (36.7 GB Ultra 160/m SCSI)	328
IBM	Deskstar 60GXP (60.0 GB ATA-100)	303
Maxtor	DiamondMax Plus D740X (80 GB ATA-133)	301
Seagate	Barracuda ATA IV (80 GB ATA-100)	296
Quantum	Fireball Plus AS (60.0 GB ATA-100)	295
Quantum	Atlas V (36.7 GB Ultra160/m SCSI)	269
Seagate	Barracuda 180 (180 GB Ultra160/m SCSI)	249
Maxtor	DiamondMax 536DX (100 GB ATA-100)	248
Seagate	Barracuda 36ES (36 GB Ultra160/m SCSI)	222
Seagate	U6 (80 GB ATA-100)	210
Samsung	SpinPoint P20 (40.0 GB ATA-100)	192

ZD Business Disk WinMark 99 in MB/Sec

Manufacturer	Model	MB/second
Seagate	Cheetah X15-36LP (36.7 GB Ultra160/m SCSI)	13.1
Maxtor	Atlas 10k III (73 GB Ultra160/m SCSI)	12.0
IBM	Ultrastar 36Z15 (36.7 GB Ultra160/m SCSI)	11.3
Fujitsu	MAM3367 (36 GB Ultra160/m SCSI)	11.1
Seagate	Cheetah 36ES (36 GB Ultra160/m SCSI)	10.5
Seagate	Cheetah 73LP (73.4 GB Ultra160/m SCSI)	10.2
Seagate	Cheetah 36XL (36.7 GB Ultra 160/m SCSI)	9.9
Western Digital	Caviar WD1000BB-SE (100 GB ATA-100)	9.8
Fujitsu	MAN3735 (73 GB Ultra160/m SCSI)	9.1
Western Digital	Caviar WD1200BB (120 GB ATA-100)	8.9
IBM	Deskstar 60GXP (60.0 GB ATA-100)	8.8
Seagate	Barracuda ATA IV (80 GB ATA-100)	8.5
Maxtor	DiamondMax Plus D740X (80 GB ATA-133)	8.0
Quantum	Atlas V (36.7 GB Ultra160/m SCSI)	7.9
Quantum	Fireball Plus AS (60.0 GB ATA-100)	7.7
Seagate	Barracuda 36ES (36 GB Ultra160/m SCSI)	7.4
Seagate	Barracuda 180 (180 GB Ultra160/m SCSI)	7.1
Maxtor	DiamondMax 536DX (100 GB ATA-100)	6.9
Samsung	SpinPoint P20 (40.0 GB ATA-100)	6.5
Seagate	U6 (80 GB ATA-100)	6.3

The file and web server benchmarks (also available at [Ra01]) are not useful since they include 80 and 100 per cent read accesses, which is not really typical of MTA servers.

5.2.3 Performance of Berkeley DB

Some preliminary, very simple performance tests with Berkeley DB 4.0.14 have been made. Two benchmark programs have been used: bench_001 and bench_002 which use Btree and Queue as access methods. They are based on `examples_c/bench_001.c` that comes with Berkeley DB. Notice: the access method Queue requires fixed size records and the access methods is record numbers (simply increasing). This method may be used for the backup of the incoming EDB. Notice: the tests have not (yet) been run multiple times, at least not systematically. Testing showed that the runtimes may vary noticable. However, the data can be used to show some trends.

Possible parameters are:

-n N	number of records to write
-T N	use transactions, synchronize after N transactions
-l N	length of data part
-C N	do a checkpoint every N actions and possibly remove logfile

Unless otherwise noted, the following tests have been performed on system 1, see Section 5.2.1. Number of records is 100000 unless otherwise noted, t/s is transactions (records written) per second.

Vary synchronization (-T):

Prg	-T	-l	real	user	sys	KB/s	t/s
1	100000	20	14.73	5.99	1.00	132	6788
1	10000	20	14.64	5.85	1.29	133	6830
1	1000	20	18.14	6.02	1.10	107	5512
1	100	20	70.57	6.03	1.76	27	1417
2	100000	20	11.58	2.91	0.74	168	8635
2	10000	20	10.14	2.86	0.85	192	9861
2	1000	20	11.20	2.85	0.95	174	8928
2	100	20	68.71	2.73	1.61	28	1455

Vary data length, first program only:

Prg	-T	-l	real	user	sys	KB/s	t/s
1	100000	20	14.39	5.93	1.16	135	6949
1	10000	20	16.77	5.91	1.16	116	5963
1	1000	20	16.58	5.91	1.13	117	6031
1	100	20	68.10	5.95	1.85	28	1468
1	100000	100	23.30	5.57	1.90	419	4291
1	10000	100	30.56	5.56	1.90	319	3272
1	1000	100	33.39	5.51	1.99	292	2994
1	100	100	82.58	5.47	2.62	118	1210
1	100000	512	96.03	7.69	4.78	520	1041
1	10000	512	94.12	7.39	5.03	531	1062
1	1000	512	97.67	7.20	5.15	511	1023
1	100	512	164.13	7.51	5.67	304	609
1	100000	1024	304.88	10.88	10.62	327	327
1	10000	1024	270.00	10.69	10.66	370	370
1	1000	1024	275.27	10.91	11.06	363	363
1	100	1024	346.10	11.01	12.09	288	288
1	100000	2048	788.88	22.18	27.59	253	126

The test has been aborted at this point. Maybe run it again later on.

Vary data length, second program only:

Prg	-T	-l	real	user	sys	KB/s	t/s
2	100000	20	9.46	2.81	0.80	206	10570
2	10000	20	11.53	2.88	0.81	169	8673
2	1000	20	12.47	2.83	0.96	156	8019
2	100	20	67.91	2.80	1.59	28	1472
2	100000	100	13.57	2.92	1.20	719	7369
2	10000	100	18.62	3.07	1.17	524	5370
2	1000	100	19.04	2.92	1.20	512	5252
2	100	100	72.73	2.80	2.16	134	1374
2	100000	512	46.10	3.90	2.61	1084	2169
2	10000	512	53.55	3.84	2.79	933	1867
2	1000	512	66.71	3.65	3.05	749	1499
2	100	512	105.25	3.36	3.76	475	950
2	100000	1024	103.72	4.92	4.68	964	964
2	10000	1024	105.53	4.87	4.82	947	947
2	1000	1024	105.60	4.73	4.85	946	946
2	100	1024	145.14	4.73	5.84	688	688
2	100000	2048	194.70	7.44	8.09	1027	513
2	10000	2048	197.09	7.22	8.15	1014	507
2	1000	2048	200.09	7.10	8.70	999	499
2	100	2048	234.85	6.86	9.53	851	425

Put the directory for logfiles on a different disk (/extra/home/ca/tmp/db), using Btree.

Prg	-T	-l	real	user	sys	KB/s	t/s
1	100000	20	14.90	6.05	0.96	131	6711
1	10000	20	14.46	5.95	1.12	135	6915
1	1000	20	17.70	5.83	1.08	110	5649
1	100	20	63.91	5.92	1.74	30	1564
1	100000	100	27.00	5.53	1.90	361	3703
1	10000	100	33.39	5.63	1.92	292	2994
1	1000	100	29.16	5.63	1.75	334	3429
1	100	100	72.18	5.44	2.42	135	1385
1	100000	512	96.94	7.49	5.09	515	1031
1	10000	512	107.99	7.34	5.17	463	926
1	1000	512	97.05	7.21	5.54	515	1030
1	100	512	145.15	7.85	5.36	344	688
1	100000	1024	268.88	10.67	11.54	371	371
1	10000	1024	279.65	11.02	11.05	357	357
1	1000	1024	304.07	10.58	11.69	328	328
1	100	1024	319.74	10.88	12.10	312	312
1	100000	2048	738.38	23.07	27.13	270	135
1	10000	2048	651.86	22.70	26.92	306	153
1	1000	2048	693.13	21.79	28.63	288	144
1	100	2048	724.68	22.51	29.04	275	137

Put the directory for logfiles on a different disk (/extra/home/ca/tmp/db), using Queue.

Prg	-T	-l	real	user	sys	KB/s	t/s
2	100000	20	10.92	2.90	0.65	178	9157
2	10000	20	9.94	2.87	0.77	196	10060
2	1000	20	31.66	2.85	0.88	61	3158
2	100	20	60.74	2.93	1.36	32	1646
2	100000	100	13.62	3.09	0.95	717	7342
2	10000	100	19.30	3.02	1.17	505	5181
2	1000	100	15.55	3.16	1.08	628	6430
2	100	100	71.88	2.97	1.72	135	1391
2	100000	512	52.08	3.93	2.50	960	1920
2	10000	512	52.42	3.68	3.03	953	1907
2	1000	512	56.58	3.91	2.90	883	1767
2	100	512	95.38	3.74	3.64	524	1048
2	100000	1024	107.20	4.69	4.87	932	932
2	10000	1024	100.15	4.88	4.57	998	998
2	1000	1024	100.95	4.78	5.06	990	990
2	100	1024	139.38	4.71	5.61	717	717
2	100000	2048	187.78	7.68	8.41	1065	532
2	10000	2048	189.76	7.09	8.62	1053	526
2	1000	2048	201.95	7.37	8.65	990	495
2	100	2048	217.66	7.21	9.53	918	459

Machine 2b: Vary data length, first program:

Prg	-T	-l	real	user	sys	KB/s	t/s
1	100000	20	21.56	9.04	1.88	90	4638
1	10000	20	13.02	9.58	1.92	150	7680
1	1000	20	12.64	9.40	1.81	154	7911
1	100	20	16.35	9.68	1.73	119	6116
1	100000	100	32.79	9.16	4.60	297	3049
1	10000	100	25.05	9.54	4.11	389	3992
1	1000	100	23.69	9.80	4.39	412	4221
1	100	100	28.51	10.25	3.89	342	3507
1	100000	512	47.67	13.82	13.65	1048	2097
1	10000	512	48.04	13.22	13.64	1040	2081
1	1000	512	46.35	13.16	14.54	1078	2157
1	100	512	52.10	13.78	11.93	959	1919
1	100000	1024	109.32	21.59	25.00	914	914
1	10000	1024	107.94	19.97	26.49	926	926
1	1000	1024	108.74	20.13	26.06	919	919
1	100	1024	113.14	20.01	26.45	883	883
1	100000	2048	240.16	44.55	55.72	832	416
1	10000	2048	262.05	43.58	54.94	763	381
1	1000	2048	245.93	41.17	57.54	813	406
1	100	2048	254.97	41.39	59.63	784	392

Vary data length, second program:

Prg	-T	-l	real	user	sys	KB/s	t/s
2	100000	20	9.85	5.92	1.30	198	10152
2	10000	20	7.82	5.90	1.28	249	12787
2	1000	20	7.21	5.13	1.34	270	13869
2	100	20	10.36	5.79	1.23	188	9652
2	100000	100	10.22	5.84	2.73	955	9784
2	10000	100	10.54	6.11	2.72	926	9487
2	1000	100	10.68	6.12	2.40	914	9363
2	100	100	13.57	6.06	2.37	719	7369
2	100000	512	23.73	7.32	8.89	2107	4214
2	10000	512	25.36	7.42	8.44	1971	3943
2	1000	512	26.12	7.19	8.56	1914	3828
2	100	512	33.79	7.24	8.78	1479	2959
2	100000	1024	47.93	9.05	12.29	2086	2086
2	10000	1024	52.26	9.63	14.91	1913	1913
2	1000	1024	52.07	9.37	14.50	1920	1920
2	100	1024	58.91	9.49	14.52	1697	1697
2	100000	2048	74.59	15.42	20.55	2681	1340
2	10000	2048	72.47	14.99	21.50	2759	1379
2	1000	2048	78.38	14.54	21.93	2551	1275
2	100	2048	76.63	14.01	22.12	2609	1304

Machine 7: Vary data length, second program only; the times for a second test run are added on the right, these clearly show how wildly the results can vary.

Prg	-T	-l	real	user	sys	KB/s	t/s	real	user	sys
2	100000	20	5.20	2.00	0.30	375	19230	5.0	2.0	0.3
2	10000	20	6.20	2.00	0.30	315	16129	4.8	1.9	0.3
2	1000	20	7.10	2.00	0.30	275	14084	7.8	2.0	0.3
2	100	20	25.80	2.00	0.50	75	3875	28.5	2.0	0.5
2	100000	100	6.30	2.10	0.60	1550	15873	6.0	2.0	0.6
2	10000	100	6.50	2.10	0.60	1502	15384	7.6	2.1	0.6
2	1000	100	10.60	2.20	0.60	921	9433	11.5	2.0	0.6
2	100	100	36.50	2.10	0.80	267	2739	31.4	2.1	0.8
2	100000	512	33.40	2.70	2.80	1497	2994	18.5	2.6	2.0
2	10000	512	29.80	2.70	2.90	1677	3355	24.6	2.5	2.3
2	1000	512	29.30	2.60	2.50	1706	3412	32.9	2.5	2.8
2	100	512	65.90	2.60	2.90	758	1517	61.0	2.5	2.8
2	100000	1024	50.50	3.30	4.90	1980	1980	58.4	3.2	5.5
2	10000	1024	60.80	3.40	5.40	1644	1644	47.2	3.2	4.6
2	1000	1024	51.70	3.30	4.60	1934	1934	45.1	3.2	4.2
2	100	1024	89.70	3.20	5.60	1114	1114	82.0	3.2	4.9
2	100000	2048	90.20	4.40	8.90	2217	1108	86.9	4.3	9.1
2	10000	2048	92.80	4.30	9.10	2155	1077	67.1	4.3	7.3
2	1000	2048	93.50	4.60	7.80	2139	1069	66.0	4.2	7.0
2	100	2048	134.00	4.40	7.50	1492	746	107.7	4.2	6.5

Vary data length, first program only:

Prg	-T	-l	real	user	sys	KB/s	t/s
1	100000	20	6.90	3.10	0.40	283	14492
1	10000	20	7.20	3.30	0.50	271	13888
1	1000	20	9.90	3.30	0.50	197	10101
1	100	20	28.90	3.20	0.60	67	3460
1	100000	100	11.30	3.40	1.00	864	8849
1	10000	100	12.20	3.30	1.00	800	8196
1	1000	100	14.00	3.30	1.10	697	7142
1	100	100	35.80	3.30	1.30	272	2793
1	100000	512	37.10	4.50	4.20	1347	2695
1	10000	512	50.00	4.60	4.50	1000	2000
1	1000	512	62.50	4.50	4.60	800	1600
1	100	512	68.60	4.50	4.60	728	1457
1	100000	1024	86.20	6.20	8.70	1160	1160
1	10000	1024	117.10	6.00	8.40	853	853
1	1000	1024	78.90	6.10	7.80	1267	1267
1	100	1024	109.60	6.10	7.40	912	912
1	100000	2048	225.80	10.90	15.90	885	442
1	10000	2048	259.40	10.80	16.30	771	385
1	1000	2048	382.60	10.90	17.40	522	261
1	100	2048	394.30	10.90	17.20	507	253

Machine 10a:

Prg	-T	-l	real	user	sys	KB/s	t/s
1	100000	20	5.00	4.40	0.50	390	20000
1	10000	20	5.00	4.30	0.60	390	20000
1	1000	20	5.50	4.40	0.80	355	18181
1	100	20	9.00	4.50	3.90	217	11111
1	100000	100	6.10	4.70	1.20	1600	16393
1	10000	100	6.20	4.80	1.20	1575	16129
1	1000	100	6.70	4.60	1.80	1457	14925
1	100	100	10.90	5.00	4.30	895	9174
1	100000	512	13.30	6.50	5.10	3759	7518
1	10000	512	12.90	6.90	4.80	3875	7751
1	1000	512	14.00	7.00	5.00	3571	7142
1	100	512	19.00	7.10	8.40	2631	5263
1	100000	1024	19.70	8.80	8.40	5076	5076
1	10000	1024	19.30	9.20	8.20	5181	5181
1	1000	1024	19.90	9.20	8.70	5025	5025
1	100	1024	26.70	9.20	12.30	3745	3745
1	100000	2048	32.90	13.80	11.70	6079	3039
1	10000	2048	31.10	13.80	12.10	6430	3215
1	1000	2048	34.90	14.40	12.30	5730	2865
1	100	2048	41.30	14.10	16.10	4842	2421

Prg	-T	-l	real	user	sys	KB/s	t/s
2	100000	20	4.70	4.20	0.30	415	21276
2	10000	20	4.70	4.00	0.50	415	21276
2	1000	20	5.20	4.20	0.70	375	19230
2	100	20	8.80	4.10	3.90	221	11363
2	100000	100	5.50	4.30	0.80	1775	18181
2	10000	100	5.70	4.30	0.80	1713	17543
2	1000	100	6.20	4.50	1.00	1575	16129
2	100	100	9.70	4.50	4.20	1006	10309
2	100000	512	12.50	5.50	2.30	4000	8000
2	10000	512	13.60	5.40	2.60	3676	7352
2	1000	512	11.70	5.10	3.30	4273	8547
2	100	512	14.50	5.70	6.40	3448	6896
2	100000	1024	17.90	6.80	3.90	5586	5586
2	10000	1024	17.30	6.70	4.60	5780	5780
2	1000	1024	18.40	6.60	4.60	5434	5434
2	100	1024	19.00	7.00	8.10	5263	5263
2	100000	2048	24.80	8.80	6.90	8064	4032
2	10000	2048	21.20	9.00	6.80	9433	4716
2	1000	2048	20.90	9.10	7.20	9569	4784
2	100	2048	24.00	8.90	11.30	8333	4166

General notice: the benchmark programs have been run while the machines were "in use", so some unusual results can be explained by the activity of other processes.

Comments:

- Compared with fsseq3 (see Section 5.2.1), Berkeley DB has to write at least two files, the logfile itself and the database. Hence the throughput should be lower by at least a factor of 2. Additional disk head movements cause another slowdown.
- For 1 it doesn't help much to put the directory for logfiles on a different disk, which is a bit surprising.
- The throughput for Queue is about 30 to 200 per cent higher than for Btree. Since the former has only fixed record size and (almost) no search overhead, this is expected. However, it is not clear whether the Queue access method is flexible enough for the deferred EDB. Currently it doesn't seems so.

5.3 Miscellaneous about Performance

2004-03-26 Effect of logging using logging to files via sm I/O.

On FreeBSD 4.9, UFS, softupdates, SCSI, smX.0.0.12, relay 5000 messages, 100 threads:

logging	time
same disk, smioerr	137-141
same disk, smioout	104
RAM, smioerr	104

This means there is a performance hit of about 35 per cent if `smioerr` is used instead of `smioout`. The former uses line buffering, hence there are more writes involved.

5.4 Performance of Various Programs

5.4.1 TCP/IP Performance

The program `checks/t-net-0.c` can be used for very simple performance test of local TCP/IP (`AF_INET` or `AF_LOCAL`) communication. This function uses the SM I/O layer on top of sockets. Some of the options which are available are listed below:

-b n	set buffer size to n, default 8192
-c n	act as client, write n bytes
-s n	act as server, read n bytes
-R n	read and write n times
-u	use a Unix domain socket

The numbers reference the machines listed in Section 5.2.1.1.

1:

-R	-c	time INET	time LOCAL
100000	32	15	10
100000	64	19	12
100000	128	24	17
100000	256	31	25
100000	512	49	43
100000	1024	84	81

7:

-R	-c	time INET	time LOCAL
100000	32	10	7
100000	64	11	7
100000	128	13	9
100000	256	17	14
100000	512	23	20
100000	1024	37	35

8:

-R	-c	time INET	time LOCAL
100000	32	51	41
100000	64	57	46
100000	128	66	60
100000	256	97	86
100000	512	148	138
100000	1024	250	243

9:

-R	-c	time INET	time LOCAL
100000	32	67	52
100000	64	74	59
100000	128	85	71
100000	256	114	97
100000	512	159	148
100000	1024	263	246

11:

-R	-c	time INET	time LOCAL
100000	32	99	89
100000	64	108	94
100000	128	138	115
100000	256	141	151
100000	512	199	221
100000	1024	346	373

Notice: these times vary wildly since the machine is used by several people.

13:

-R	-c	time INET	time LOCAL
100000	32	46	38
100000	64	59	48
100000	128	78	70
100000	256	120	110
100000	512	203	192
100000	1024	376	358

5.4.2 DB Lookup Performance

Just a preliminary number: On system 1 the example program `examples_c/bench_001.c` achieves about 1 to 1.5 millions lookups per second (this is for a data length of 20 bytes and a cache size of 64MB, which more or less means direct memory access, no disk I/O). Taking the results from 5.4.1 into account means that this is factor of 100 faster than performing lookups over a generic TCP/IP connection. This certainly must be taken into account for the decision how and where to incorporate DB lookups.

Using larger data sizes (256 to 512 bytes) and smaller caches (10000 bytes) cause a significant drop in performance: a sequential lookup of all data varies from 60000 to 10000 lookups per second (on system 1).

Random access goes down as much as 1000 to 2000 lookups per second.

5.4.3 `snprintf` Performance

On AIX, `sm_snprintf()` is about 2 times slower than `snprintf()`. On SunOS 5.8 it's about 1.3, on FreeBSD there is no difference (which isn't surprising since it's almost the same code). It might make sense to use the native `snprintf()` version on some platforms, however, this isn't possible anymore due to the extensions in `sm_snprintf()` (which supports more format specifiers, e.g., for constant strings).

Bibliography

[ABLL92] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Transactions on Computer Systems,*, volume 10(1), pages 53–79. ACM, February 1992.

[ASF] Apache. `http://www.apache.org/`.

[Ber] Dan Bernstein. Secure interprocess communication. `http://cr.yp.to/docs/secureipc.html`.

[Ber97] Dan Bernstein. Verp: Variable envelope return paths, 1997. `http://cr.yp.to/proto/verp.txt`.

[Ber98] Dan Bernstein. qmail, 1998. `http://cr.yp.to/qmail.html`.

[Bra89] Requirements for internet hosts – application and support. RFC 1123, Internet Engineering Task Force, 1989.

[Dre02] Dreppner. http://people.redhat.com/drepper/nptl-design.pdf, 2002. `http://people.redhat.com/drepper/nptl-design.pdf`.

[FB96] N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. RFC 2045, Internet Engineering Task Force, 1996.

[Fre96] N. Freed. SMTP service extension for returning enhanced error codes. RFC 2034, Internet Engineering Task Force, 1996.

[Fre00] N. Freed. SMTP service extension for command pipelining. RFC 2920, Internet Engineering Task Force, 2000.

[GK98] R. Gellens and J. Klensin. Message submission. RFC 2476, Internet Engineering Task Force, 1998.

[Haz01] Philip Hazel. Exim, 2001. `http://www.exim.org/`.

[Hof99] P. Hoffman. SMTP service extension for secure SMTP over TLS. RFC 2487, Internet Engineering Task Force, 1999.

[ine] inetd. `src/usr.sbin/inetd/inetd.c`.

[ISC01] ISC. Bind, 2001. `http://www.isc.org/bind.html`.

[Keg01] Dan Kegel. The C10K problem, 2001. `http://www.kegel.com/c10k.html`.

[KFM95] J. Klensin, N. Freed, and K. Moore. SMTP service extension for message size declaration. RFC 1870, Internet Engineering Task Force, 1995.

[KFR+94] J. Klensin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. SMTP service extension for 8bit-mimetransport. RFC 1652, Internet Engineering Task Force, 1994.

[KFR+95] J. Klensin, N. Freed, M. Rose, E. Stefferud, and D. Crocker. SMTP service extensions. RFC 1869, Internet Engineering Task Force, 1995.

[Kle01] Simple mail transfer protocol. RFC 2821, Internet Engineering Task Force, 2001.

[Krü98] Jan Krüger. check local, 1998. `http://www.digitalanswers.org/check_local`.

[LDA] Openldap. `http://www.OpenLDAP.org/`.

[Moo96a] K. Moore. An extensible message format for delivery status notifications. RFC 1894, Internet Engineering Task Force, 1996.

[Moo96b] K. Moore. SMTP service extension for delivery status notifications. RFC 1891, Internet Engineering Task Force, 1996.

[Mye96] John Myers. Local mail transfer protocol. RFC 2033, Internet Engineering Task Force, 1996.

[Mye99] J. Myers. SMTP service extension for authentication. RFC 2554, Internet Engineering Task Force, 1999.

[New00] D. Newman. Deliver by SMTP service extension. RFC 2852, Internet Engineering Task Force, 2000.

[NGP02] Next generation posix threading project, 2002. `http://oss.software.ibm.com/pthreads/`.

[nss] nsswitch.conf. `nsswitch.conf(2)`.

[OBS99] Mike Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *USENIX Annual Technical Conference*. The USENIX Association, June 1999.

[Par86] Craig Partridge. Mail routing and the domain system. RFC 974, Internet Engineering Task Force, 1986.

[pf] pf. `pf.conf(5)`.

[pro] procmail. `http://www.procmail.org/`.

[Ra01] Eugene Ra. Storagereview.com's testbed3, 2001. `http://www.storagereview.com/articles/200111/20011109R`

[RB01] Jeffrey B. Rothman and John Buckman. Which OS is fastest for High-Performance Network Applications?, 2001. `http://www.sysadminmag.com/newsletters/feature/`.

[Res01] Internet message format. RFC 2822, Internet Engineering Task Force, 2001.

[RO92] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured filesystem. In *ACM Transactions on Computer Systems,*, volume 10(1), pages 26–52. ACM, February 1992.

[SBMS93] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. The design and implementation of the 4.4bsd log-structured file system. In *Proceedings of the 1993 Winter Usenix, San Diego, CA,*. The USENIX Association, January 1993.

[SGI01] SGI. State threads for internet applications, 2001. `http://state-threads.sourceforge.net/`.

[She01a] Gene Shekhtman. Re: some questions, 2001. `http://oss.sgi.com/projects/state-threads/`.

[She01b] Gene Shekhtman. St and dbms client libraries, 2001. `http://oss.sgi.com/projects/state-threads/`.

[Slea] Sleepycat. Berkeley db 4.0.14 change log. `http://www.sleepycat.com/update/4.0.14/if.4.0.14.html`.

[Sleb] Sleepycat. Berkeley db tutorial and reference guide, version 4.1.24. `db-4.1.24.NC/docs/ref/`.

[Sol02] Multithreading in the Solaris operating environment. Technical report, Sun Microsystems, 2002.

[SSB+95] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 Winter Usenix,*, pages 249–264. The USENIX Association, January 1995.

[Ste92] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[Ste98] Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice Hall, 2nd edition, 1998.

[SV96a] Douglas C. Schmidt and Steve Vinoski. Object interconnections: Comparing alternative programming techniques for multi-threaded corba servers (column 6). *SIGS C++ Report*, April 1996.

[SV96b] Douglas C. Schmidt and Steve Vinoski. Object interconnections: Comparing alternative programming techniques for multi-threaded corba servers (column 7). *SIGS C++ Report*, July 1996.

[SV96c] Douglas C. Schmidt and Steve Vinoski. Object interconnections: Comparing alternative programming techniques for multi-threaded servers (column 5). *SIGS C++ Report*, February 1996.

[Var01] Sam Varshavchik. Courier mta, 2001. `http://www.courier-mta.org/`.

[Vau96a] G. Vaudreuil. Enhanced mail system status codes. RFC 1893, Internet Engineering Task Force, 1996.

[Vau96b] G. Vaudreuil. The multipart/report content type for the reporting of mail system administrative messages. RFC 1892, Internet Engineering Task Force, 1996.

[Ven98] Wietse Venema. postfix, 1998. `http://www.postfix.org/`.

[Win96] J. De Winter. SMTP service extension for remote message queue starting. RFC 1985, Internet Engineering Task Force, 1996.