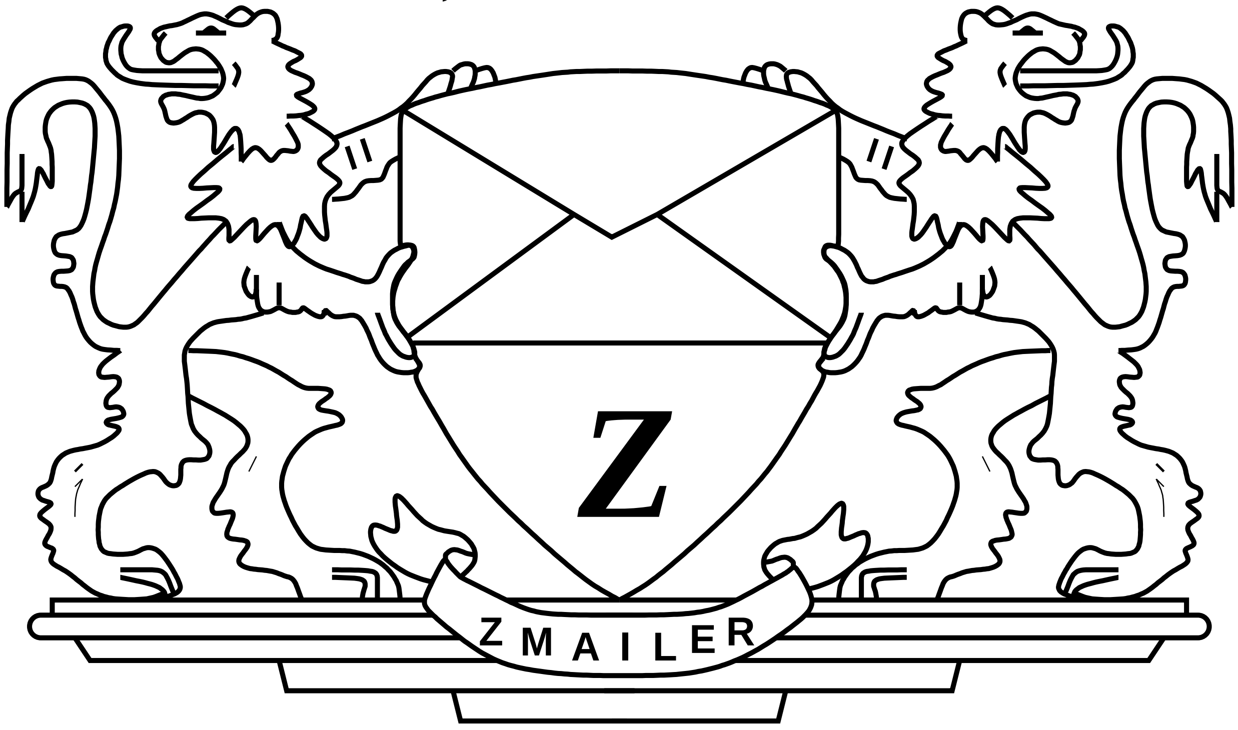
ZMailer

The Manual; v1.99.26.200602020935



Matti Aarnio

ZMailer: The Manual; v1.99.26.200602020935

Document SGML source rendered to PDF on: Thu Feb 2 09:35:27 UTC 2006

Warning

THIS IS A WORK IN PROGRESS, AND ALL OF THE ORIGINAL LATEX MATERIAL HAS NOT BEEN YET CONVERTED TO DOCBOOK SGML!

Once full conversion has been achieved, a lot of information updates are also needed.

Warning

In the meantime, if some module or subsystem does have incomplete documentation, see if man-pages are more complete!

by Matti Aarnio

Table of Contents

List of Figures

I. Tutorial

Chapter 1. Introduction to Email

1.1. Messaging

This chapter is quite different from the rest of this document. Here we build a foundation for understanding messaging, instead of focusing on how ZMailer behaves.

This chapter may feel a bit theoretical and abstract, being detached from practical life.

In reality, however, experience shows that most problems with messaging are a result of not understanding the underlying messaging model, or of not respecting said model.

The terminology used here may seem a bit X.400 oriented. It is, because folks from what was then known as CCITT (now known as ITU-T) adapted the model originally developed by IFIP. Of course, CCITT added a lot of things of its own invention (like ADMDs and PRMDs), that we don't need to bother ourselves with.

Although the terminology comes from X.400, it is in no way restricted to it. Our presentation here is a generic messaging presentation not restricted to any type of protocol.

Messaging, as the name says, is all about exchanging messages, short (or sometimes long) pieces of information. Messaging is always directional (which means that there is always a sender and one or more recipients), targeted (the list of recipients is fixed) and store-and-forward based.

There are a few messaging-like applications in which the message is broadcast to a wide, unspecified audience. A prime example of this latter application is Usenet News. News is not messaging, as it is not targeted.

So what about mailing lists that are linked to News, are they messaging or not? As long as the message is transported as a mail message, it is messaging. One of the recipients of the message may well be a Usenet News newsgroup. Similarly, a sender of a message might be the News system, or the author who initiated the submission by using News. Messaging is not necessarily interpersonal.

It is also quite normal for different applications to communicate by using messaging methods. A prime example of this would be EDI traffic. It is clearly messaging, but not interpersonal.

1.1.1. The Messaging Model

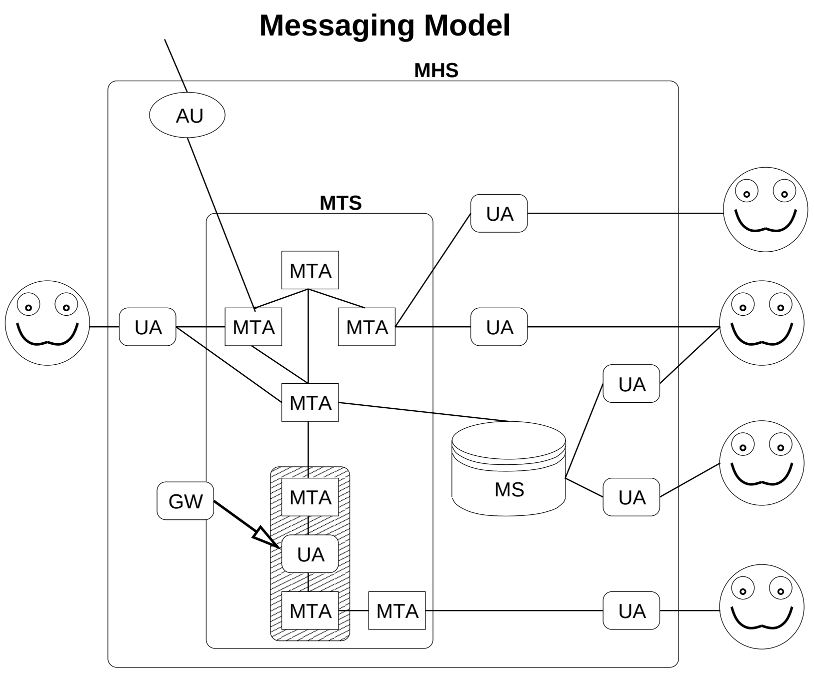
In addition to users, the basic building blocks of messaging are *User Agents (UAs)* and *Message Submission Agent (MSA)* is newer term for specific sub-task of Internet email, namely authenticated message submission to first MTA of MTS system. *Message Transfer Agents (MTAs)*. User agents are the interface through which a human user interacts with the messaging system. On non-interpersonal messaging user agents may be built directly into applications. MTAs are used to transport messages from one computer system to another. An example of a good MTA is ZMailer. *Access Units (AUs)* can be used for accessing telematic services, for example telefax. (Or in general act on user behalf somehow, e.g. automated scripts.) *Message Stores (MSs)* can be used between MTAs and UAs. They are used for storing messages before and while UAs are used to access and manipulate them. *Message Delivery Agent (MDA)* is newer term intended to cover specific sub-task of moving the message from MTA system to the care of MS, UA or AU. Some new MTA suites even do parts of the UA functionality ("`.forward`" -processing) in the MDA. *Gateways (GWs)* are used in between two different types of messaging systems, for example between SMTP and X.400 mail. Gateways are conceptually made of two MTAs and a UA that interconnects them.

There are two more acronyms worth looking at in the messaging model, namely *MTS (Message Transport System)* and *MHS (Message Handling System)*. MTS is the collection of MTAs (and GWs and MSAs and MDAs), while MHS includes MTS and UA functionality (UAs, MSs and AUs).

All in all, X.400 terms are not a complete match on how things are done in Internet email, nor should they be considered as normative, merely giving you a reasonable frame of reference that isn't very wrong.

A graphical example of the messaging model is shown in figure Figure 1-1. It shows the relationships between different elements of the model.

Figure 1-1. A graphical example of the messaging model.



As can be seen, a user may use more than one UA, and a UA can be connected to more than one MTA.

Although it is important to understand the relationships between different entities in the messaging model, it is even more important to understand the nature of a message and the way UAs and MTAs deal with it.

A message consists of a body and headers. In case of messages with more than one body-part (for example some MIME messages) the different body parts are all part of the outermost body-part.

Figure 1-2. How a message looks normally.

```
Headers
  From:    The Manager
  To:      One Bright Employee
  CC:      secretary
  Subject: Salary raise
  Date:    17 May 1997

Body
  Dear Employee,
  ...
  The Manager
```

Figure 1-3. A possible, more complex message structure.

```
Headers
Body
  Headers
  Body
    Headers
    Body
    Headers
    Body
  Headers
  Body
```

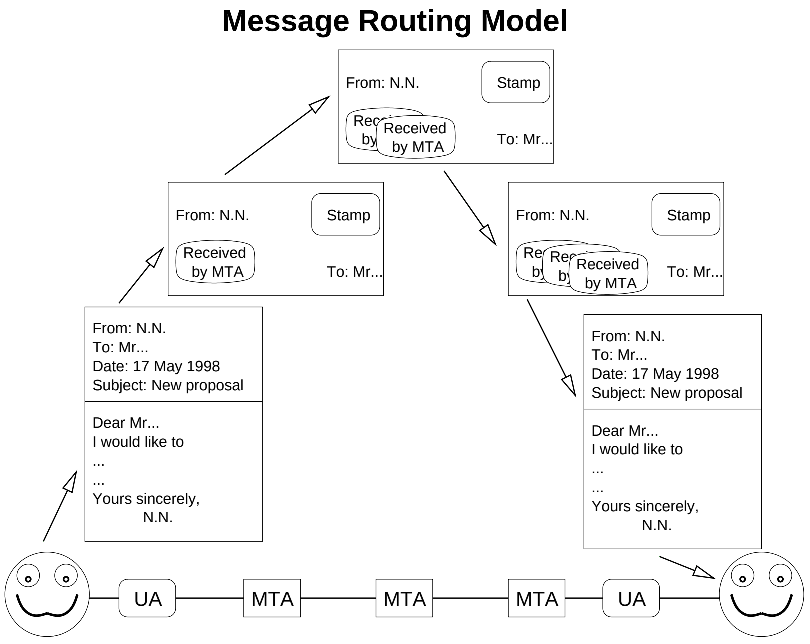
As can be seen, there is always just one outermost body that contains all other body-parts. In some cases, for example X.400 the protocol seems to violate this by leaving out the outermost body-part. However, even on those cases we must assume, at the abstract level, that the outermost body-part is there.

However, this is not all that there is to the structure of a message. When a message is in transit, being handled by MTAs it is put inside an envelope, just like a normal letter is inside an envelope while the postal service is carrying it. Just like the postal service is not permitted to look inside the envelope, neither are MTAs permitted to look inside. Whenever there is a need to look inside the envelope, it is always a UA function, and done on behalf, and on the authority of, a UA.

There are some violations of this. When using the SMTP protocol, the `Received:` lines are put in the headers by MTAs. This is bad engineering, but as the process of adding a new header line is fairly straightforward, it doesn't cause too much pain. In some cases MTAs, and especially the GWs modify the header even more, and sometimes they even mess with the body. This is a sure recipe for trouble.

Graphically, the way a message should be dealt with is shown in figure Figure 1-4.

Figure 1-4. How a message should be handled.



The user creates the message with the help of a UA. How the interaction is arranged is a local matter. Once the message has been prepared, it is passed to a nearby MTA together with necessary envelope information and put into an envelope. The MTA puts its stamp on the envelope to show that it has received the message. The first MTA passes the message to the second MTA. The second MTA puts its stamp to the message and passes it to the third MTA, and so on. The final MTA passes the message to a UA, and the envelope is removed.

There are at least three ways to pass the message from an MTA to a UA. The message may be pushed to a (running) UA, a UA may pull it from an MTA, or an MTA may pass the message to an MS from which a UA will receive it at a convenient time.

The normal UNIX way of delivering mail (`/usr/spool/mail/user`) can be seen as any of the above three mechanisms, but should normally be seen as a UA pulling a message. The reason for this confusion is with the de-facto SMTP standard MTA, Sendmail.

Although Sendmail is in many ways a very clever piece of software, it is also the reason for many problems, as it has blurred the line between MTA and UA. Sendmail is clearly an MTA, but it also performs many of the UA level functions, like handling of "`.forward`" files. This behaviour has become the de-facto standard way for UNIX MTAs to behave, making it necessary for any MTA, including ZMailer, to behave the same way.

1.1.2. Routing And Delivering Messages

As MTAs only deal with envelopes, all routing and delivery decisions have to be based on information available on the envelope. It follows from this that the envelope and headers may contain conflicting information. This is normal, and is not a cause for worry.

MTAs may, and often do modify addresses present in the envelope. This might include changing addresses to a format more suitable for mail delivery and alias expansion.

It is important to make a distinction between aliasing and forwarding mail. Aliasing is an MTA function, in which an MTA effectively knows that to reach a seemingly local user, mail should be sent to a different address. To accomplish this, the MTA changes the recipient information on the envelope. Forwarding is a UA function. When forwarding, the mail message is received by the original, intended recipient, and re-sent to another address. Although forwarding is a UA function, it doesn't have to result in a change to body or headers, but on the envelope, both sender and recipient should be changed. Sendmail bluntly violates this, and makes most other MTAs violate it as well.

Most mailing lists today are just alias expansions, on which one recipient address on the envelope is replaced with multiple addresses. In many cases this is a reasonable approach. However, all major mailing lists should be set up as a UA function. This involves changing not only the recipient address but also the sender address in the envelope. In this way, undeliverable messages are sent to the owner of the list, who can deal with the problem, and not to the sender of the message, who can do nothing to remedy the situation. (The ZMailer has some built-in facilities for this, see about *Mailing Lists...* on section Section 13.2.2.)

Error messages must be sent to the envelope sender address, as this is a MTA function.

Replies to messages should be sent to header addresses, because replying is a UA function.

Chapter 2. ZMailer's Features, and Facilities

2.1. Introduction

ZMailer is a mailer subsystem for the UNIX operating systems. It is in charge of handling all mail messages that are created on a system, from their creation until final disposition locally or by transfer to another system.

As such, the mailer subsystem (the Message Transfer Agent) must interface to local mail reading and composing programs (User Agents), to the various transport methods that can be used to reach other mailers, and to a variety of databases describing the mailer's environment.

ZMailer provides this functionality in a package and with a philosophy that has benefitted from experiences with earlier mailers.

ZMailer provides a capable, robust, efficient subsystem to do the job, which will excel in demanding environments, but is technologically simple enough to fit easily everywhere. (In UNIX-like environments.)

However ZMailer is not smallest memory footprint MTA there is, nor it tries to be. What it has and does, are ways to limit resource expenditure, while still providing high-performance services. One can't very easily overwhelm a machine where ZMailer MTA runs by just feeding it too much work in parallel. (Message Queues can grow beyond any reason, but even there are safety limiters.) Limits are available for example:

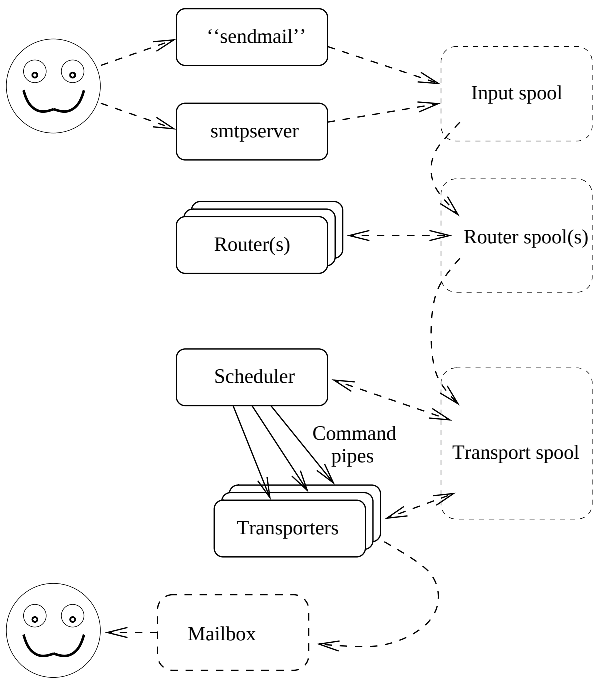
- Number of parallel incoming SMTP sessions overall
- Number of parallel incoming SMTP sessions from any given single IP address
- Number of messages per source IP address in timeframe
- Number of parallel internal processing programs
- Number of parallel SMTP sessions feeding messages outside

For ZMailer's incoming SMTP interaction there are ways to define, that the usual anonymous user from given address space can send only so many messages with only so many recipients per time interval. Such limits help keeping Zombie botnet Windows machines from causing too much trouble. Technology aimed for keeping such service abusers under control is ever evolving, and can probably never be 100% capable. Also the abusers do learn rather quickly what is bad in their behaviour, and they do modify their programs to get past any filters thrown at them. At the same time, legitimate users are hadly ever evolving their behaviour, and are always behaving rather foolishly, or rather their used UA softwares behaves more like hijacked Zombies, or perhaps there is just a NAT box, and several users beyond it...

In the longer run, service providers will need bigger and bigger servers (or clusters of smaller servers) to make their inbound and outbound SMTP services. They will also need ways to track user behaviour online, and if necessary, modify reaction algorithms in timescales of few hours. Further complication is, that the service providers need different behaviour models according to "this is Consumer" vs. "this is Corporate" customer. Corporate clients usually have some MTA of their own, and thus their legitimate network behaviour is a lot more like what spammers do these days, and separating spammer hijacked system there is really difficult.

2.1.1. Design Summary

Figure 2-1. ZMailer's processes.



ZMailer is a multi-process mailer, using three daemon processes to manipulate messages. Used technologies are as simple as possible, e.g. while networking stuff is as advanced as possible (with fallbacks to simplest basic behaviour), some of other modern things (like threads) are not used in favor of simpler approaches

- Message arrive in via **sendmail** program "API" for internally originated messages, or via **Smtpserver** subsystem, which is system front-door for messages coming in from the external network. The **Smtpserver** is in reality a cluster of auxiliary programs providing efficient low-overhead support for various analysis things needed while messages are coming in. Second one of these processes is a **Router**, and makes all decisions about what should happen to a message; routing and possibly message header visible things rewriting. The third daemon is a message queue manager, **Scheduler**, used to schedule the delivery of messages. The Router uses a configuration file that closely follows *Bourne shell script* syntax and semantics, with minimal magic. Message files are moved around in a series of directories, and the **Scheduler** and its **Transport Agents** run off of control files created by the **Router**.
- The **Sendmail** is very simple "plug-compatible" message submission agent for system internal message submissions into the ZMailer MTA, and does all its things without any sort of set-uid privilege escalation needs.
- The **Smtpserver** has evolved into rather complicated animal, as it aims to really efficiently support things that in the early days required truly heavy-weight auxiliary program startups for every incoming SMTP connection -- startup of such auxiliaries has now been shared over a large number of arriving connection and messages by means of having them as **Smtpserver's** permant auxiliary helpers:
  - First is a slightly less burdened **Router** for determining if source or destination address domains are possibly known in the system, and primarily being able to reject messages at front

  door that are destined to nonexistent addresses. (This instance is separate from *main* **Router**, and thus is somewhat duplicating main Router's task, but this doesn't e.g. do list expansions and other such expensive things.)

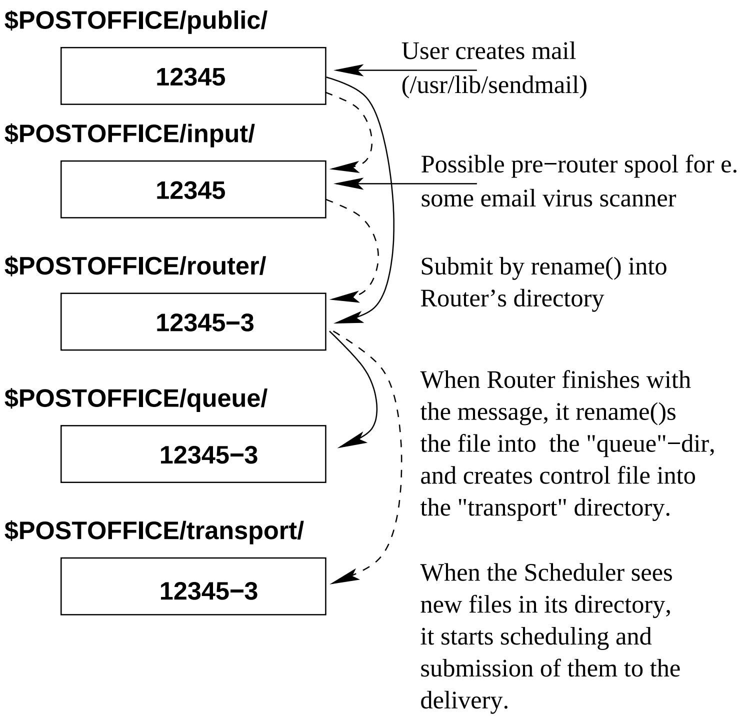
  - There is also **Content-Filter** that can be used up to how ever complicated message content analysis *syncronously* with incoming message feed (there is also a possible "input" subdirectory for offline non-synchronous content analysis, however synchronous processing has certain appeal in itself, not the least the ability to tell message sender to go and stuff their SPAMs into...)
  - And third is the **Rate-Tracker** subsystem, that can keep track of such things as "Non-authenticated customer at IP address N.N.N.N has sent more than 60 messages in past hour, stop that sending until the sliding window allows more to be sent."

  These auxiliary servers do operate so that when smtpserver-subsystem shutdown has been ordered, and last client needing support goes away, they drop away themselves.

- The *main **Router** subsystem* will process messages one at a time (per **Router** instance), as it finds them in a directory where *User Agents* submit their outgoing messages. *Message Envelope* and *Message Header* information is all kept in the same *message file* along with the *message body*, and this file is never modified by any ZMailer program. After parsing the envelope and RFC822 header information, the **Router** validates the information extracted, and calls functions defined in the configuration file to decide exactly, how to deliver the message and how to transform the embedded addresses. The algorithms that do this are easily re-configurable, since the control flow and address manipulation is specified by familiar(ish) shell script statements. When the **Router** is finished on a message, it will produce a *message control file* for use by the delivery processing stage of ZMailer, and move the original message file to another location.
- Once the main **Router** subsystem has decided what to do with each of the addresses in a message, the **Scheduler** builds a summary of this information by reading the control file created by the **Router**. This knowledge is merged with a data structure it maintains that stores which messages are supposed to be sent where, and how. According to a pre-arranged agenda, the **Scheduler** will execute delivery programs to properly move the message envelope, header, and body, to the immediate destination. These delivery programs are called *Transport Agents*, and communicate with the **Scheduler** using a simple protocol that tells them which messages to process and returns status reports to the **Scheduler**. The **Scheduler** also manages status reports, taking appropriate action on delivery errors and when all delivery instructions for a message have been processed.
- There are several standard *Transport Agents* included with the ZMailer distribution. The collection currently includes a local delivery program, an SMTP client implementation, and a Transport Agent that can run Sendmail-M-line-compatible delivery programs.
- A separate **mailq** utility allows querying the **Scheduler** for the state of its queues. For existing Sendmail installations, a replacement program is included that simulates most of the Sendmail functionality in the ZMailer environment. This allows ZMailer to replace a **Sendmail** installation without requiring changes in standard *User Agents*.
- Several other tools and utilities exist for other specific purposes.
- There are also facilities that allow *loosely coupled cluster* creation. Each member machine of the cluster is separate entity, but they can keep track of *rate-tracking data* of all of their neighbours, as well as activate on demand *ETRN queue flush* in cluster wide setup. None of these is guaranteed in any strict way, which makes their implementation technology considerably simpler, and also occasional limits that they implement are not considered to be upheld rigorously under every possible conditions. A tripple-sigma reliability on e.g. limit enforcement is considered "a plenty good enough".

In loosely coupled clusters, *delivery to system internal (UNIX-style) message store is not clusterized*, unless the store in question is some sort of e.g. LMTP connected external Message Store. (Even UNIX-style mailbox files may work, if underneath there is some sort of cluster-wide filesystem, which has working FCNTL locks.) One such way was once to use front-line processing to map arrived message to actual recipient and node into which it was expected to go, and to proxy POP and IMAP services so that users didn't need to know on which node their mailboxes did actually reside.

Figure 2-2. Directories that ZMailer uses for message processing.


$POSTOFFICE/public/
12345
User creates mail (/usr/lib/sendmail)
$POSTOFFICE/input/
12345
Possible pre−router spool for e. some email virus scanner
$POSTOFFICE/router/
12345−3
Submit by rename() into Router's directory
$POSTOFFICE/queue/
12345−3
When Router finishes with the message, it rename()s the file into the "queue"−dir, and creates control file into the "transport" directory.
$POSTOFFICE/transport/
12345−3
When the Scheduler sees new files in its directory, it starts scheduling and submission of them to the delivery.

2.2. Running ZMailer

ZMailer is fairly simple to run, once the setups are completed it can be left to run on its own with very little supervision.

Things that might need supervision are things like:

- Timely cycling of log files, which otherwise will grow until they fill all of the available disk space (One need not log everything possible, about the only thing this system does not allow you to log is the message body content.)
- Keeping watchful eye on `$POSTOFFICE/freezer/`, and `$POSTOFFICE/postman/` directories. Former for processing SPAM email, latter for pathological problem cases.

  "Logging and Statistic for the Administrator:" Chapter 16, "Checking the Log Files:" Section 4.6, "Trim-down of Logging:" Section 4.8, "Postmaster Analysis Area:" Section 11.4.

We look closer into these issues at latter parts of this document, but now it is sufficient to tell, that the principal tool for active monitoring of the system health is command:

```
$ mailq -ss
```

which does tell, if router, or scheduler are up and about, or not, and also does tell about the sizes of the different sub-spools.

The general management interface for starting and stopping different subsystems is command

```
# zmailer
```

which the system installs into `$MAILBIN/` directory, and which command usually needs a symlink to itself from some more common location for administrative convenience (`/usr/sbin/zmailer` -> `$MAILBIN/zmailer`) so that the administrator does not need to add `$MAILBIN/` directory into his or her PATH. On overall, it is intention that not even admin user should need to run directly the programs located at the `$MAILBIN/` directory.

Basically the administration is as follows:

- At system start-up (to start all subsystems):

  ```
  # zmailer
  ```

- At system shutdown (to kill all subsystems):

  ```
  # zmailer kill
  ```

There is also a way to make sure the system will not let the ZMailer to start at the system start-up, because you have some massive work going on, and the system is not in condition to accept email for a while:

```
# zmailer freeze
```

and the antidote for the "freeze" is, naturally:

```
# zmailer thaw
```

Normal operations can not be started at "frozen" system without "thawing" it at first.

The user-visible component of the ZMailer is (for de-facto interface)

```
$ /usr/lib/sendmail
```

(a.k.a. `/usr/sbin/sendmail`) which is "simple" message submission program that mimics sendmail commands behaviour, but of course many details of sendmail are not really implemented at all, mostly because they do not have equivalents in the ZMailer system.

There are also functional equivalents (or near equivalents) of other sendmail/system utilities: **mailq**, **newaliases**, and **vacation**.

2.3. Factors Affecting Overall System Performance

- Speed of `$POSTOFFICE/` directory filesystem. Specifically directory metadata operation speed (e.g. fully synchronous directory metadata update is way slower, and safer, than fully asynchronous.)
- Possible separate filesystem spindle on `$POSTOFFICE/transport//` directory.
- Amount of memory, and thus filesystem buffering, and (lack of) swapping by component processes.
- DNS server quality
- "Staticness" of routing data, e.g. the less there is DNS lookups involved for email delivery, the better it works.

II. Build and Install

This section describes how to build and install ZMailer.

> **Tip:** Consider joining the ZMailer user-community email list, or at least reading/searching its archive. It is the place to meet the Gurus, in case you have problems. See the “`Overview`” file in the source distribution for more information.

Chapter 3. Build and Install

3.1. Environment Issues

The cornerstone of everything in busy Internet email routing is a well-working DNS server, and modern resolver library. If you use the BIND name-server, you should be using (or install) a recent version, As of this writing (January 2006), BIND server developers recommend version 9. They also strongly recommend, that you do not let zone data masters (either masters, or slave copies) to do any recursive resolving, and do recursive resolvings with servers that do not have locally mastered data.

For performance reasons you should have *local* instance of recursively resolving caching name-server. (And for *security reasons* it should not do any local DNS zone masterings.)

You may also want to support any of following facilities by pre-installing them into your system (before compiling and installing ZMailer, that is):

- OpenSSL for in- and outbound encrypted SMTP sessions
- TCP-Wrapper (can also do without it)
- LDAP client library (if necessary)
- Private replacement library for `getpwnam`(), and/or for `zgetpwnam`()
- Whoson service to tie in with e.g. POP and IMAP servers for authenticating SMTP relaying. (But the use of SMTP authentication is definitely preferred instead of using external hacks like "POP-before-Post".)

3.2. Auto-configuration

The `zmailer.conf` file carries various so called “ZMailer Environment” configuration variables.

In following we refer to those often in style of:

```
$MAILVAR   $MAILSHARE   ...
```

Which essentially means expanding in Bourne-shell like manner given *ZENV-variable* from this file.

The location of this file is defined at system configuration, originally it was at: `$MAILSHARE/zmailer.conf`, but these days more often at the `$MAILVAR/`.

This system uses several preferably separate partitions for different things:

- Software binaries, and databases: `$MAILBIN/` (site shared, read-only), `$MAILSHARE/` (site shared, read-only), `$MAILVAR/` (node local db’s, r/w)
- The mailbox spool: `$MAILBOX/` (`/var/mail`)
- The postoffice spool: `$POSTOFFICE/` (`/var/spool/postoffice/`)
- The log directory: `$LOGDIR/` (`/var/log/mail`)

Important: *A filesystem without following two properties is not suitable for ZMailer's* `$POSTOFFICE/`:

- *Files must succeed to be* `link`*(2):ed in between directories within the filesystem without copying them.*
- *The file i-node numbers must not change with* `rename`*(2) or* `link`*(2) calls applied to them.*

Most of the `$POSTOFFICE/` directory must be a single mounted filesystem within which files can be `link`(2)ed from one directory to another, as well as moved around with `rename`(2)

However, the `$POSTOFFICE/transport/` subdirectory *can be* separate filesystem mounted under `$POSTOFFICE/`! Such arrangement can (under some situations) result in additional system performance, as transport agents need to modify (write locks) files in the `$POSTOFFICE/transport/` subdirectory, while they only read (without locks) files in `$POSTOFFICE/queue/` subdirectory.

Adding system reliability in form of having directory data committed to disk at the time of the directory modifying operations returning with success is a nice bonus, although in normal UFS-like cases that taxes system performance heavily.

E.g. running fast-and-loose with async metadata updates in Linux EXT2 filesystem gives you performance, but in case the system crashes, your postoffice directory may be in shambles, and important email may have been lost.

How exactly you can combat the problem is yours to choose. Most filesystems for UNIX have lots of different options at mount-time, and also by-directory attributes can be set to control these things. Check yours after your decide on what kind of data loss threat you can tolerate at the expence of what speed reduction. (E.g. 300+ day straight uptime with power surges during a thunderstorm at the end of it toasting your machine along with its disks and filesystems, but trouble-free running until then ?)

The GNU autoconf mechanism is used, however, you still may need to touch on some files after that system has run through: You MUST define `--prefix=` so that ZMailer components end up in reasonable places. The `$MAILBIN/` (and `$MAILSHARE/`, and `$MAILVAR/`) variable values are derived from the `--prefix=`, which can cause surprises if you do **make install** with GNU autoconf defaults.

When choosing your prefix, do try to keep is fairly short, as there are a few scripts which concatenate string-components of:

```
"#! "+prefix+"/bin/router -f"
```

and usually systems have a limit of 32 characters for that, which gives at most 15 characters for your prefix!

Also, if the `$MAILSHARE/zmailer.conf` file exists[1], it is read to initialize several different environment paths (including `$MAILBIN/`, et.al.!)

```
# ./configure                            \
  --prefix=/opt/mail                     \
  --with-postoffice=/var/spool/postoffice \
  --with-mailbox=/var/mail               \
  --with-logdir=/var/log/mail
```

Or an example from my development machine:

```
$ ./configure --prefix=/opt/mail
```

```
creating cache ./config.cache
***
*** You can set  ZCONFIG  environment variable to define
*** the location of the (default) /opt/mail/zmailer.conf -file
*** (You can use also   --with-zconfig=  -parameter)
***
*** Consider also setting following parameters:
***   --mandir=DIR     -- for man-pages
***   --libdir=DIR     -- for libzmailer(3)
***   --includedir=DIR -- for libzmailer(3)
*** (They can be outside the --prefix=DIR -tree)
***
*** You can set CC, and CFLAGS  environment variables to
*** choose the C-compiler, and its options, especially at
*** systems where there are multiple choices to use...
***
```

You can also go into a sub-directory, and configure and compile there: (But it may need GNU make as system "**make**"!)

```
$ mkdir myhost ; cd myhost
$ ../configure ...
$ make ...
```

See if `SiteConfig` makes sense now, if not, you can tune most of the values with various `--with-*=` keywords:

```
$ ./configure --help
```

Explanations about these configuration options are listed at chapter Chapter 6.

Those options that you can't tune, you can edit at the "`SiteConfig.in`" file. (Redo the configure with new parameters, if that looks to be necessary approach.)

Additional examples:

- DEC OSF/1 at nic.funet.fi with DECs best compiler...

```
$ CFLAGS="-O -g3 -std1" CC="cc -migrate" \
  ./configure --prefix=/l/mail
```

- Sun Solaris 2.5 at mailhost.utu.fi, SunSoft CC

```
$ CC="cc -O" ./configure --prefix=/opt/mail
```

- Sun Solaris 2.5 at mailhost.utu.fi, gcc-2.7.2

```
$ ./configure --prefix=/opt/mail --with-gcc
```

- Sun Solaris 8 at marsu.funet.fi, gcc-2.95.2 (egcs-1.1.2)

```
$ 'CC='gcc' CFLAGS='-g -O'  ./configure    \
      --prefix=/opt/mail --with-ipv6        \
      --with-zconfig=/opt/mail/zmailer.conf
```

- Linux-2.0.x/libc-5.4.2 at mea.cc.utu.fi, gcc-2.7.2

```
$ ./configure --prefix=/l/mail
```

3.3. Compilation

At the top-level, run

```
$ make
```

or perhaps:

```
$ make clean all
```

which at first cleans everything, and then makes — great if you changed some configuration parameters.

This should compile everything, and leave a `zmailer.Config` file in the top-level directory. Nothing outside the source area will be touched at this point.

(If your system "**make**" lets your shell "`SHELL`" environment affect its own execution environment, it may be that non-**sh/ksh/zsh** users detect weird phenomena, and failures. Beware!)

3.4. Installing and Upgrading

This section describes how to install or upgrade ZMailer.

3.4.1. Install Preparation

If you are currently running a zmailer, kill off all mailer processes using

```
$ zmailer kill
```

and save the state of your system. This includes any active contents of the `$POSTOFFICE/`, as well as database files and anything else in the installation areas you want to be sure to keep. This is just paranoia, the installation should not overwrite precious files, and will save old versions of distribution files in "`bak/`" sub-directories.

The interface in between the commonly used sendmail, and ZMailer is a "compatibility program", which is to replace the `/usr/lib/sendmail` (a.k.a. `/usr/sbin/sendmail` on some systems). The system attempts to automate the replacement, but it *may* present a cry for help if your system does not have functioning symlinks. Also if "**test -h $SENDMAILPATH**" does fault in mysterious ways, the reason may be that your system does not have symlinks.

If you are currently running Sendmail, kill your SMTP server and drain the Sendmail queue. There is no automatic method to re-queue Sendmail messages under ZMailer. If you later want to back out to Sendmail, all you need to do is move the former version of the sendmail (on `/usr/lib/sendmail.bak`, for example) binary back to `/usr/lib/sendmail`.

(You may also need to do some magics with system start-up scripts in case you are running SysV-style init. BSD `/etc/rc.local` does need its own gymnastics too. Sample SysV-init script is at file: "`proto/zmailer.init.sh`")

A sort of method to quickly handle your sendmail queue is to start ZMailer's SMTP server, reconfigure the old sendmail to use smart-host, which happens to be at the same machine. (Or at an adjacent machine if you moved the queue, or ...) Anyway the point is to get the sendmail to send its queue via SMTP to the ZMailer. An equally valid option is to drain the sendmail's queue by running sendmail in queue drain mode only, although `/usr/lib/sendmail` points to ZMailer's "**sendmail**."

3.4.2. Installation

Once you are safe, run:

```
# make install
# $MAILBIN/post-install -MD5
```

*(Substitute **$MAILBIN** with the path where your binaries go)*

This installs all binaries and the default configuration and database files, as well as creates `$POSTOFFICE/` directories. The configurations will still need editing! See below.

The **post-install** handles important activity in tracking the base versions of configuration files by storing MD5 check-sums of original files with `.md5` suffix tagged to them into their original location (`proto/` sub-directories). This way when sysadmin changes something, the new run of **post-install** will detect the changes and not write over such file.

There exists also a way to do the installation into a "parallel universe" by means of install-time "prefix" environment variable:

```
# . SiteConfig
# DESTDIR=/var/tmp/build
# make install DESTDIR=$DESTDIR
# $DESTDIR$MAILBIN/post-install --destdir $DESTDIR
```

which of course expects to have "`/var/tmp/build/`" directory in existence, and possibly some others under it, but you will soon see, what it needs. (But **post-install** does not support that, and so it needs to be used at the last stage of packaged ZMailer's installation)

3.4.3. Installing the Manual Pages.

Because for a long time the installation location of ZMailer's man-pages has not had an obvious destination location, normal "make install" run at the top-level does *not* install them!

Go into the `man/` directory, and install the manual pages by hand:

```
# cd man
# make
```

This will tell what you can order it doing, and what is the default for `MANDIR` is at the moment. If the default is right:

```
# make install
```

or in case the default guessing didn't get it right:

```
# make install MANDIR=/our/manpages
```

Notes

1. Default location is `$MAILSHARE/zmailer.conf`, and it can be changed with `--with-zconfig=` option.

Chapter 4. System Configuring

4.1. Run-time files

This section describes the configuration in short. More detailed information can be found in Administration and Reference parts.

4.1.1. `zmailer.conf`

The `zmailer.conf` file carries various so called “ZMailer Environment” configuration variables.

In following we refer to those often in style of:

```
  $MAILVAR   $MAILSHARE   ...
```

Which essentially means expanding in Bourne-shell like manner given *ZENV-variable* from this file.

The location of this file is defined at system configuration, originally it was at: `$MAILSHARE/zmailer.conf`, but these days more often at the `$MAILVAR/`.

This system uses several preferably separate partitions for different things:

- Software binaries, and databases: `$MAILBIN/` (site shared, read-only), `$MAILSHARE/` (site shared, read-only), `$MAILVAR/` (node local db’s, r/w)
- The mailbox spool: `$MAILBOX/` (`/var/mail`)
- The postoffice spool: `$POSTOFFICE/` (`/var/spool/postoffice/`)
- The log directory: `$LOGDIR/` (`/var/log/mail`)

4.1.2. `/etc/group`

The default configuration also expects to find names of trusted users listed at `/etc/group` entry `zmailer`. Users (unames) listed there will be able to claim any addresses at the message headers, etc. (See `$MAILSHARE/cf/trusted.cf` for its usage there.)

The usual *minimal* set is: `root,daemon,uucp`. (Note: At some machines “daemon” is called “daemons”; *It must be on that group for the smtpserver to be able to work properly!*)

Warning

SECURITY ITEM: Those users at `zmailer` group *must not* contain `nobody`!

The `nobody` is used to prevent externally given inputs from being able to execute arbitrary programs at the system, or from writing to arbitrary files.

4.1.3. `/etc/services`

Add the following line to `/etc/services` in the section for host-specific services:

```
mailq   174/tcp  # Mailer transport queue
```

Indeed this isn't *quite* mandatory, as the **scheduler** subsystem can be configured to use different IPC socket. See more about this at Scheduler's PARAM entries: Section 22.1.1.

4.2. The Router subsystem

4.2.1. The Router Configuration File (`$MAILSHARE/router.cf`).

You must now pick a top-level router configuration file. The default is provided in `proto/cf/SMTP.cf(.in)`. (The **post-install** places it into `$MAILSHARE/router.cf`).

You need to *verify* `$MAILSHARE/router.cf`.

Some real-life samples of `router.cf` are at the `proto/` directory in the source tree.

4.2.2. `$MAILVAR/mail.conf`

If you are using the default configuration setup, the `router.cf` file expects to find a `$MAILVAR/mail.conf` file containing three variable definitions:

```
  # Where am I?
  orgdomain=domain
  # Who am I?
  hostname=host.subdomain.$orgdomain
  # Who do I claim to be?
  mydomain=subdomain.$orgdomain
```

For example:

```
  orgdomain=toronto.edu
  hostname=relay.cs.$orgdomain
  mydomain=cs.$orgdomain
```

Create `$MAILVAR/mail.conf` with appropriate contents. If you are a multi-host site, determining these things can be automated according to your local policies and conventions. See the files specific to the University of Toronto (`UT*.cf`) for examples of this.

Location of this file is written in `$MAILSHARE/router.cf`. By editing that entry you can alter it.

Caution

Note that “hostname=” entry above *is not alone sufficient* for getting the system to know all of the domains it should consider as local. See below about `localnames` database.

4.2.3. Verifying That the Router Starts

At this point, you should be able to start the **router** process in interactive mode. Run:

```
# $MAILBIN/router -i
```

or

```
# /usr/lib/sendmail -bt
```

You should see something like:

```
ZMailer router (2.99.55 #4: Tue Feb 22 15:24:09 EET 2001)
you@hostname:/some/path/to/src/zmailer/router
Copyright 1992 Rayan S. Zachariassen
Copyright 1992-2001 Matti Aarnio
Configured with command: 'CC='gcc -Wall' CFLAGS='-g -O' ../configure &hellip;'

z#
```

If there are errors in the configuration file, you will be told here. The “z#” is the interactive prompt for root. It is unlikely you can do anything useful before setting up the data files, so get out of this by hitting EOF, or type **`exit`**.

4.2.4. The Router Database Files

Now you should merge, replace, and very least check the default database and forms files against your previous setup.

In older systems users had fixed choices on which databases to have at the **router** subsystem for which looked up relation. Newer systems have `$MAILBAR/db/dbases.conf` configuration file to tell the association of database to look-up relations, and also the source files.

4.2.4.1. `$MAILBIN/zmailer` script

You may want to add a symbolic link from some directory in your path to `$MAILBIN/zmailer`, if you don’t already have this. I put this link in `/usr/local/sbin`.

This script is skeleton driver for lots of things in the ZMailer, including the not so small a feat of acting as SysV-Init’s init-script as well.

4.2.4.2. `$MAILVAR/db/dbases.conf` file

As mentioned above, this file configures (when exists) database look-up functions versus actual back-end databases.

With this file one can have e.g. multiple *aliases* databases which are bound together in pre-determined query order (first matcher wins).

This configuration file is used to translate a semi-vague idea about what database sources (in what forms) are mapped together under which look-up names, and what format they are, etc...

This is used by "**zmailer newdb**" command to generate all databases described here, and to produce relevant `.zmsh` scripts for the **router** to use things. The "**zmailer newdb**" invocation does not mandate **router** restart in case the database definitions have not changed; reverse is true: If definitions are added/modified/removed, the **router** *must* be restarted!

*When you use "**zmailer newdb**" command, you recompile all databases defined in the controlling configuration file `$MAILVAR/dbases.conf` -- or if you don't have that file, then all databases listed below, as creation/update of those are the system defaults.*

For more complete example, see the default boilerplate version of this file.

Figure 4-1. Sample of "`$MAILVAR/db/dbases.conf`" file

```
#|Fields:
#|     relation-name
#|         dbtype(,subtype)
#|              dbpriv control data (or "-")
#|                  newdb_compile_options (-a for aliases!)
#|                     dbfile (or "-")
#|                         dbflags (or "-") ...
#|
#| The  dbtype  can be "magic" '$DBTYPE', or any other valid database
#| type for the Router.  Somewhat magic treatment (newdb runs) are
#| done when the dbtype is any of: *DBTYPE/dbm/gdbm/ndbm/btree
#|
#| The "dbfile" need not be located underneath of $MAILVAR, as long as
#| it is in system local file-system (for performance reasons.)  E.g.
#| one can place one of e.g. aliases files to some persons directory.
#|

aliases     $DBTYPE 0:0:644    -la $MAILVAR/db/aliases     -lm
aliases     $DBTYPE majordomo:0:644 -la /opt/Majordomo/md-aliases -lm

fqdnaliases $DBTYPE root:0:644 -la $MAILVAR/db/fqdnaliases -lm%
userdb      $DBTYPE root:0:644 -la $MAILVAR/db/userdb      -lm

routesdb    $DBTYPE -    -l  $MAILVAR/db/routes      -lm% -d pathalias
thishost    $DBTYPE -    -l  $MAILVAR/db/localnames  -lm  -d pathalias
```

4.2.4.3. `$MAILVAR/db/aliases` file

The provided skeleton aliases file on purpose contains syntax errors, so you are reminded to change the contents.

Choose one of the following methods to rebuild the database:

```
# $MAILBIN/newaliases
# $MAILBIN/zmailer newaliases
# /usr/lib/sendmail -bi
# /usr/bin/newaliases
# $MAILBIN/zmailer newdb
```

If there are errors, correct them in the "`aliases`" file, and repeat the command until the alias database has been initialized. The final message should look something like:

```
 319 aliases, longest 209 bytes, 16695 bytes total
```

exact numbers vary, of course...

See also IETF's *RFC 2142: "Mailbox Names for Common Services, Roles and Functions"* (http://www.ietf.org/rfc/rfc2142.txt) (file `doc/rfc/rfc2142.txt`) for other suggested aliases you may need.

4.2.4.3.1. Alias expansion

Read the notes on alias expansion in the file `doc/guides/aliases` and on mailing list maintenance in Section 13.2.2, *Mailing Lists and `~/.forward`*.

4.2.4.4. `$MAILVAR/db/fqdnaliases` file

The `fqdnaliases` database is for mapping fully-qualified user addresses to others — for example your machine has a set of domain-names for it to consider local, but you want to have separate people to be postmasters for each of them as shown at Figure 4-2.

Figure 4-2. Sample of "`fqdnaliases`" file

```
postmaster@domain1: person1
postmaster@domain2: person2
postmaster@domain3: person3, person4
```

It is also possible to shunt all recipient addresses for given domain to some arbitrary addresses as shown at Figure 4-3.

Figure 4-3. Second sample of "`fqdnaliases`" file

```
@domain4:  person4
@domain5:  %1@domain6
```

This facility is always in stand-by — just add the file, and you have it at the next router start-up.

The "**`%1`**" local part is special (and experimental, as of 21-Feb-2001) substitution pattern where *local part (user)* can be replaced into the looked up data. More details at Section 21.5.16 (dblookup), and at Section 21.5.50 (`relation` declaration).

You may even handle just a few users for each of those domains, and then have the "`routes`" entry (see below at Figure 4-5) to declare something suitable:

```
.domain1  error!nosuchuser
.domain1  error!nosuchuser!%0
```

which combined with the "`fqdnalias`" method will let "`postmaster@domain1`" to exist, and report error on all others.

Choose one of the following methods to rebuild the database:

```
# $MAILBIN/newfqdnaliases
```

or either of:

```
# $MAILBIN/zmailer newfqdnaliases
# $MAILBIN/zmailer newdb
```

If there are errors, correct them in the "`fqdnaliases`" file, and repeat the command until the alias database has been initialized. The final message looks similar to that of the ordinary aliases case.

If you have multiple fqdnaliases databases defined at the `dbases.conf`, you must use the "**zmailer newdb**". (See Section 24.1.8.)

4.2.4.5. `$MAILVAR/db/localnames` file

Add the host-names you want ZMailer to do local delivery for, to the `$MAILVAR/db/localnames` file. Due to my own belief in Murphy, I usually add partially qualified domain names and nicknames in addition to canonized names. If you want to do local delivery for mail clients, put their names in here too. You may use pathalias style "*.domain*" names in this file, to indicate everything under some subdomain.

With the sample config files for ZMailer-2.98, and latter, this `localnames` is actually a mapping of those various names to the desired forms of the canonical name, thus an example as seen in figure Figure 4-4.

Figure 4-4. Sample of "`localnames`" file

```
#
# Left:  input name
# Right: what is wanted to be shown out
#
# List here all names for the system
#
astro.utu.fi         astro.utu.fi
oj287                astro.utu.fi
oj287.astro.utu.fi   oj287.astro.utu.fi
oj287.utu.fi         astro.utu.fi
sirius               sirius.utu.fi
sirius.astro.utu.fi  sirius.utu.fi
sirius.utu.fi        sirius.utu.fi
```

In certain cases the **router** is able to deduce some of the names, *however smtpserver anti-relay policy compiler will not be able to do so, and needs this data!*

THUS: All names that the host may ever have are best listed in here! It reminds you of them, and makes sure a message destined into the host really is accepted.

Compile this into run-time binary database with command:

```
# zmailer newdb
```

(fall-back method is sequential re-scan of the text file)

4.2.4.6. `$MAILVAR/db/routes` file

Add any UUCP neighbours or other special cases to this file. For an example see Figure 4-5.

You can compile the file into binary database with command:

```
# zmailer newdb
```

Figure 4-5. Sample of "`routes`" file

```
#
# "routes" mapping file
#
.toronto.ca       error!err.wrongname
.toronto.cdn      error!err.wrongname
alberta           uucp!alberta
atina             smtp![140.191.2.2]
calgary           smtp!cs-sun-fsa.cpsc.ucalgary.ca
icnucevm.bitnet   smtp!icnucevm.cnuce.cnr.it
```

4.2.4.7. UUCP Node Names

If your hostname and UUCP node name are not identical, put your UUCP node name in the file `/etc/name.uucp` (or `/etc/uucpname`).

4.2.5. Checking the Routing

At this point, you should be able to start the router again in interactive mode, and ask it to route addresses. Try either of:

```
# /usr/lib/sendmail -bt
# $MAILBIN/router -i
```

at the prompt:

```
z# router you
```

(where "you" is your login-id, naturally) should print out:

```
(((local you you default_attributes)))
```

Keep playing around with various addresses until you get a feel for it. Modify the configuration file if your setup requires it.

To give more feeling of what goes on during the "**route**"-command, you can give command "**rtrace**" before trying to use "**route**."

4.3. The Smtpserver subsystem

The **smtpserver** implements RFC-821 server along with lots of latter extensions.

Configurable subsystems are:

- Generic server parametrization with "`smtpserver.conf`" file.
- Relaying policy control via "`smtp-policy`" database.
- Optional message content analysis via "contentfilter" mechanism.
- Optional PAM authentication framework for SMTP AUTH extension.
- Optional externally driven program to autenticate users; command line contains username, and STDIN gets the user supplied password.

See the Administration Chapter 12 for further details.

4.3.1. The "`smtpserver.conf`", and smtp-policy databases

These take care of such a things as preventing relay-hijack type of abuse of your system.

Basically you want to install the boilerplates and the tool scripts, edit them a bit, and run **policy-builder.sh** script. For further details on this, see chapter Section 12.2.

In **smtpserver** front you may need to lower the strict standards of the basic RFC-821 SMTP protocol and allow acceptance of non-qualified addresses — ones without any sort of domain name in them.

Another thing to allow is (sigh) MS-Windows-CE 1.0/2.0 gadgets with their totally broken SMTP sending system.

Both of these things are handled by "EHLO-style options" described at chapter Section 12.1.2.

4.3.2. Testing smtpserver operationality

The **smtpserver** can be tested fully with fairly simple method -- as long as input databases are readable by the test runner:

```
$ $MAILBIN/smtpserver -i -d 1 -T '[1.2.3.4]'
$ $MAILBIN/smtpserver -i -d 1 -T '[ipv6.11::33]'
```

Above the bracketed dotted decimal address literal is source address used at policy function testing, and one should vary there systems which *are* allowed to relay thru the server, and also systems which are not allowed to relay thru the server.

Do testing by issuing normal SMTP protocol transactions, and observing the results:

```
000- Lots of debug information
...
220 some greeting
EHLO foobar
000- Lots of debug information
...
250-local.host.name Hello foobar
250-8BITMIME
250-PIPELINING
```

```
...
250 HELP
MAIL FROM:<>
000- Lots of debug information
...
250 Ok ...
RCPT TO:<user@some.where>
000- Lots of debug information
...
250 Ok ...
```

If you want to do testing without excessive amount of debug information, do leave out “-d 1” part of the start arguments.

4.4. The Scheduler subsystem

4.4.1. Checking the Scheduler

The location of the `scheduler.conf` on running system is `$MAILSHARE/scheduler.conf`

For normal operations of the system the current sample of “`scheduler.conf`” file is quite sufficient, but in case you want to do something unusual, like using procmail for local delivery, do read on.

The default “`scheduler.conf`” contains also linkage to “`scheduler.auth`” (see Section 22.3), which is access-control for interacting with the scheduler from external programs, like **mailq**.

In Figure 4-6 there are some salient points about tuning the “`local`” channel behaviour.

Figure 4-6. Sample of “`scheduler.conf`” passage for “`local/*`” selector

```
local/*
    interval=5m
    idlemax=9m
    expiry=3d
    # want 20 channel slots, but only one HOST
    maxchannel=15
    maxring=5
    # Do MIME text/plain; Quoted-Printable -> text/plain; 8BIT
    # conversion on flight!
    command="mailbox -8"
    # Or with PROCMAIL as the local delivery agent:
    #command="sm -8c $channel procm"
    # Or with CYRUS server the following might do:
    #command="sm -8c $channel cyrus"
```

There are three variants of the “`command=`” entry:

`command="mailbox -8"`

The normal ZMailer **mailbox**(8) channel program.

`command="sm -8c $channel procm"`

Variant for running procmail.

`command="sm -8c $channel cyrus"`

Variant for using CMU Cyrus message store server.

For more information regarding *scheduler* configuration language, see Section 22.1.

4.4.2. Checking `scheduler.auth` file

Access-control to the **Scheduler**'s internal state data is defined at file `scheduler.auth`, which should be usable in its default form.

For more information about this, see Administration Section 14.5, and Reference Section 22.3.

4.4.3. Checking `sm.conf` file

For some uses the **scheduler** runs **sm**(8) program — called "sendmail-like mailer".

This supports *most* of sendmail's `M`-entry flags, at least flags with versions previous to 8.11(.0)

The ZMailer **sm**(8) channel program is used to create support for things like:

- *uucp* transmits
- procmail as local delivery agent
- supporting CMU Cyrus message store as local delivery agent

For more information, see Section 23.4.1.

4.4.4. Customizing ZMailer Messages

Edit several of the canned error messages and programs (scripts) to reflect your local configuration: `$MAILSHARE/forms/` files and `$MAILBIN/ta/usenet` (injected message).

Normally the boilerplate messages looks something like these:

```
HDR From:    The Post Office <postmaster>
HDR Sender:  mailer-daemon
SUB Subject: Errors: No such user(s)
ADR Bcc:     <postmaster>

This is a collection of reports about email delivery
process concerning a message you originated:
```

In these, "ADR" lines define header lines which are to be analyzed for *recipient* addresses, while "HDR" lines can carry anything which doesn't get output as *envelope* address. The "ADR" line contained addresses *must* be in brackets, and there can be only one address per such header. If there are more, only the first one is picked.

More details at Scheduler Administration Section 14.7.1, and at Scheduler Reference Section 22.7.

4.5. Boot-up Scripts

Add something like the following lines to boot-up scripts (`/etc/rc.local` or `/etc/rc2.d/S99local` or similar):

```
if [ -r /etc/zmailer.conf ]; then
  . /etc/zmailer.conf
  if [ ${MAILSERVER-NONE} = NONE -a
       -x $MAILBIN/zmailer ]; then
    $MAILBIN/zmailer bootclean
    $MAILBIN/zmailer && (echo -n ' zmailer') >/dev/console
  fi
fi
```

For SysV-init environments, see source-tree file: `utils/zmailer.init.sh`. You may want to comment out startup of the Sendmail daemon, if you have it to begin with.

4.6. Checking the Log Files

Start ZMailer:

```
# $MAILBIN/zmailer
```

Keep an eye on the log files (`$LOGDIR/router`, `$LOGDIR/scheduler`), the `$POSTOFFICE/postman/` directory for malformed message files, and `$POSTOFFICE/deferred/` in case of resource problems.

4.7. Crontab

See Figure 4-7 for three crontab entires for the root to run. Those are:

1. This will "**resubmit**" messages that have been deferred with no useful processing possible at time of deferral. Adjust the re-submission interval to suit your environment. *Having files in "deferred" state is a sign of troubles! Always investigate!*
2. This "**cleanup**" is to regularly clean out the "`$POSTOFFICE/public/`", and "`$POSTOFFICE/postman/`" directories.
3. The automatic log-file trimmer/rotater is a good idea to have, but you need to customize it for your environment. More of that below.

You may want to hard-wire the location of the **zmailer** script.

Figure 4-7. ZMailer related crontab entries for root user

```
# Two ZMailer related root's CRONTAB entries:
28 0,8,16 * * * . /etc/zmailer.conf ; $MAILBIN/zmailer resubmit
7  4      * * * . /etc/zmailer.conf ; $MAILBIN/zmailer cleanup
```

```
FIXME!FIXME!
  # This third one will not per default be installed into your system
  0  0      * * * . /etc/zmailer.conf ; $MAILBIN/rotate-logs.sh
```

4.8. Trim-down of Logging

Once satisfied that things appear to work, you may want to trim down logging: there are four kinds of logging to deal with:

- Router logs:

  Usually kept in `$LOGDIR/router`. This is the stdout and stderr output of the router daemon. If you wish to turn it off, see `$MAILSHARE/cf/standard.cf` for routine dribble()}, and especially its invocations! Alternatively use "`-L /dev/null`" to divert everything to the "dev null."

- Scheduler logs:

  Usually kept in `$LOGDIR/scheduler`. Same as router. The **scheduler** prints there *only* when it feels bad about something.[1]

- Syslog Control ZENV variable:

  ZENV variable $`SYSLOGFLG` contains a set of single-character flags: "S", "C", "R", and/or "T". FIXME! FIXME! Explain Smtpserver/sCheduler/Router/Transport agents!

- General Mail Logs:

  Usually kept in syslog files, depending on how you have configured the syslog utility (`/etc/syslog.conf`). All ZMailer programs log using the `LOG_MAIL` facility code for normal messages. You can deal with this specifically in your **syslog** configuration file on systems with a 4.3bsd-based syslog. The following reflects the recommended configuration on SunOS 4.0:

  ```
  mail.crit    /var/log/syslog
  mail.debug   /var/log/mail/mail.syslog
  ```

  For pre-4.3bsd-based syslogs, you may want the syslog log file to be just for important messages (e.g. LOG_NOTICE and higher priority), and have a separate file for informational messages (LOG_DEBUG and up).

- By default, the postmaster will *not* receive a copy of all bounced mail; this can be turned on selectively by simply editing the various canned forms used to create the error messages. These forms are located in the `FORMSDIR` (`proto/forms` in the distribution, or `$MAILSHARE/forms` when installed). You should review these in any case to make sure the text is appropriate for your site.

Notes

1. At least this is the theory, practice may be different, though.

Chapter 5. Installation to Clients

This section describes the installation at clients.

5.1. Required Files

The following files/programs are needed on clients:

`$MAILSHARE/zmailer.conf`

The $`MAILSERVER` variable may be set to the mail server host's name. This is not required as **mailq** will usually be able to discover this by itself.

`/usr/lib/sendmail`

to submit mail

mailq

should be installed in the site's local `bin` so people can query the mail server. (Remember to update `/etc/services`)

`$POSTOFFICE/`

This directory from the server should be mounted and writable.

5.2. Mounting `$MAILBOX`es and/or `$POSTOFFICE/` Hierarchies via NFS

This is mostly for client machines, but the NFS may plaque you also at servers.

If you for some obscure reason are mounting `$MAILBOX`es and/or `$POSTOFFICE` hierarchies via NFS, do it with options to disable various attribute caches:

```
             actimeo=0
    alias:   noac
```

The best advice is to NOT to mount anything over NFS, but some people can't be persuaded...

Lots of things are done where file attributes play important role, and they are extremely important to be in sync! (Sure, the "`noac`" slows down the system, but avoids errors caused by bad attribute caches.)

If you are mounting people's home directories (`~/.forward` et. al.) via NFS, consider the same rule!

Often if the mail folder directory is shared, also one of following (depending upon the system):

```
/usr/mail
/usr/spool/mail
/var/mail
/var/spool/mail
```

Chapter 6. `./configure` options

configure options of ZMailer package, per version 2.99.55.

The configure script has three kinds of parameters for it:

- (optional) environment variables for CC="..." et.al.
- ZENV data pulled in from `$ZCONFIG` file (if it exists)
- various `--with-*` et.al. options

6.1. Used environment variables

User environment variables

`ZCONFIG`="/file/path"

Using this is alternate for using "`--with-zconfig=../`" option. Not needed if the `--prefix=` derived `$MAILSHARE/zmailer.conf` is used.

`CC`="command"
`CFLAGS`="options"

Obvious ones, compiler, and possible "-g -O" flags...

`CPPDEP`="command"

Not normally needed — builds dependencies

`INEWSBIN`=/file/path

If you want to pre-define where your "*inews*" program is — for possible use of "*usenet*" channel.

Recycled ZENV variables (from $`ZCONFIG` file):

```
For these see  SiteConfig(.in)  file

ZCONFIG=
MAILBOX=
POSTOFFICE=
MAILSHARE=
MAILVAR=
MAILBIN=
LOGDIR=
NNTPSERVER=
SCHEDULEROPTIONS=
ROUTEROPTIONS=
SMTPOPTIONS=
LOGDIR=
SENDMAILPATH=
RMAILPATH=
SELFADDRESSES=
```

6.2. Options for various facilities

Options for various facilities

`--prefix=/DIR/PATH`

The only really mandatory option, gives actually defaults for *ZENV variables*: `$MAILSHARE`, `$MAILVAR`, `$MAILBIN`.

`--with-gcc`

Compile with GCC even when you have “cc” around.

`--with-zconfig=/FILE/PATH`

Where the run-time `zmailer.conf` file is located at (and with what name). This is *the only* hard-coded info within libraries and thus programs using them. Everything else is run-time relocatable by means of using “variables” listed in this file.

Default: `$MAILSHARE/zmailer.conf`

Lots of ZCONFIG environment variables are pre-set from values present in pre-existing file.

When environment variable ZCONFIG is set and exported to the **./configure** script, use of “`--with-zconfig=no`” will set the location of the file, but prevents pre-load of various values from it to the auto-configuration environment.

`--with-zconfig-noload`

Without this option the configuration will load ZENV variable values from possibly existing ZCONFIG file.

Default: values will be preloaded without this option.

`--with-mailbox=/DIR/PATH`

Overrides system-dependent location of the user mail-boxes. Defaults are looked up thru list of directories:

```
/var/mail
/var/spool/mail
/usr/mail
/usr/spool/mail
```

First found directory will be the default — or then system yields `/usr/spool/mail`.

`--with-postoffice=/DIR/PATH`

Overrides system-dependent location of the “`$POSTOFFICE`” directory under which system stores queued email. Will try directories `/{usr,var}/spool/postoffice/` to see, if previously installed directory tree exists. Default will be `/var/spool/postoffice/` in case there is no previously created postoffice directory.

`--with-mailshare=/DIR/PATH`
`--with-mailvar=/DIR/PATH`
`--with-mailbin=/DIR/PATH`

These are overrides for values derived from `--prefix=/DIR` option, or possibly pre-loaded from ZCONFIG file.

`MAILSHARE` = "$PREFIX", `MAILVAR` = "$PREFIX", but `MAILBIN` = "$PREFIX/bin".

`--with-logdir=/DIR/PATH`

Explicit value to replace `$LOGDIR` ZENV value and/or to override default value of: `/var/log/mail/`

`--with-nntpserver=HOST`

If you want to use "*usenet*" channel, you need to name NNTP server into which you feed news with NNTP.

`--with-sendmailpath=/FILE/PATH`

Overrides for default location(s) of sendmail program. ZENV variable `$SENDMAILPATH` can be overridden with this.

`--with-rmailpath=/FILE/PATH`

Overrides for default location(s) of rmail program. ZENV variable `$RMAILPATH` can be overridden with this.

`--with-selfaddresses="NAME,NAME"`

Obsolete option regarding providing into in ZENV variable to yield system internal names auto-magically for the SMTP transport channel uses, and also for the router to see, if destination IP address is local at the system.

Usage of this option may become necessary at load-balance clusters, but even then, setting the value to the ZCONFIG file is easier than pre-configuring them.

`--with-system-malloc`

Use system `malloc`() library, don't compile own: Alternate for using: `--with-libmalloc=system` This is default.

`--with-libmalloc=LIBNAME`

Where "LIBNAME" is one of:

`system`

System `malloc`() as is.

`malloc`

Bundled "libmalloc" without debugging things.

`malloc_d`

Bundled "libmalloc" *with* debugging things.

`--with-yp`

Want to use YP, and has it at default locations

`--with-yp-lib='-L... -lyp'`

If needed to define linking-time options to find the YP-library.

`--with-ldap-prefix=DIRPREFIX`

If UMich/NetScape LDAP are available thru `DIRPREFIX/include/` and `DIRPREFIX/lib/` locations, this is a short-hand to find the interface — with files in the system primary include and lib locations, "`/usr`" is a special value which will be ignored. There is no default value for `DIRPREFIX`.

`--with-ldap-include-dir=/DIR/PATH`

Special over-rider for compilation include directory of LDAP

`--with-ldap-library-dir=/DIR/PATH`

Special over-rider for linkage library directory of LDAP

`--disable-pam`

Disable PAM(3) facility from becoming auto-configured even when its API headers and libraries are present.

`--without-fsync`

At systems where the local file-system is log-based/journaling, doing `fsync`() is wasteful. This disables `fsync`() in cases where it is not needed. (In others it may boost your system performance by about 20% — with dangers... On the other hand, once a system disk(?) fault which hang mailer at spool directory access did cause severe damage all over, and probably use of this option would not have made any difference; **fsck** was mighty unhappy.)

`--with-bundled-libresolv`

If your system is not very modern, you may consider using this option to compile in a resolver from bind-4.9.4-REL. On the other hand, if your system is modern, it may have even newer resolver in it. At such time, don't use this!

`--with-ipv6`

Use IPv6 at things where it is supported. This is often highly experimental, although many subsystems in ZMailer are built with `getnameinfo`() et.al. interfaces, which works both on IPv4 and IPv6.

`--with-ipv6-replacement-libc`

If the system needs more support for user-space IPv6 things, this generates those.

`--without-maillock`

Don't use `maillock`(3) even if system has it. (Solaris `maillock`(3) is ok, some early Linux versions weren't...)

`--without-rfc822-tabs`

Some systems dislike getting RFC-822 headers with form of:

```
"Headername: <TAB> value"
```

With this option, no TABs are used and instead "ordinary" space character is used.

In real life, this feature is superseded by the **router** to *always TABifying* all headers, and the transport-agent header write-out to *untabifying* them, if ZENV variable `RFC822TABS=` has value "0".

`--with-tcp-wrappers`
`--with-tcp-wrappers=/DIR/PATH`

Optional `=/DIR/PATH` value gives directory where there are `tcpd.h` and `libwrap.a` files. Without value this option looks for several common locations for those files, and if finds them, yields compile and linking hooks,

`--with-ta-mmap`

On some systems with good `mmap`(2) with "`MAP_FILE|MAP_SHARED`," and well behaving `munmap`() it does make sense to replace `read`()/`write`() thru a file-descriptor to the file with `mmap`() — however that requires `munmap`() not to scrub away in-mapped blocks any more actively, than the buffer-cache works at `read`()/`write`() blocks.

This was default for a while, however most systems don't have really well-behaved `munmap`()s :-/ (Perhaps IBM AIX is the only exception ?)

`--with-mboxquotacheck`

Set 'CHECK_MB_SIZE' #define for mailbox.c compilation, and expect `checkmbsize`() function to be found via `--with-generic-lib=` referred library.

`--with-privateauth"`

Use "`private/`" sub-directory in a part of **smtpserver** program compilation.

`--with-privatembox"`

Use "`private/`" sub-directory in a part of **mailbox** program compilation.

`--with-getpwnam-library="-L... -l..."`

Certain sites have expressed wishes to use their own libraries to replace the standard `getpwnam`() (and possibly `getpwuid`()) routines. These are used in **router**, **scheduler**, **mailbox**, **hold**, and **vacation** programs.

These programs use `getpwnam`() libary call to look up various customer user-names to whatever the system needs them for.

For ZMailer needs the library must support user-ids:

- root
- daemon, or daemons
- nobody

and whatever others your local system magic needs.

`--with-generic-include="-I/..."`

This parameter allows ubiquitous "`-I/...`" options to be used in all program compilations throughout the package.

`--with-generic-library="-L/..."`

This parameter allows ubiquitous "`-L/...`" options to be used in all program linkages throughout the package.

`--with-openssl`
`--with-openssl-prefix="/dir/prefix"`
`--with-openssl-include="/dir/incl"`
`--with-openssl-lib="/dir/lib"`

Search for, and use OpenSSL, if it can be found. (For optional in and outbound SMTP traffic encryption on the Internet.)

`--with-whoson`
`--with-whoson="/dir/prefix"`

This does explicit integration with "whoson" server; see the "`whoson-*.tar.gz`" file in the "`contrib/`" sub-directory.

6.3. Runtime *ZENV* Variables

Runtime *ZENV* Variables

`ZCONFIG=`

ZCONFIG is the pathname of the configuration file specifying all the other host-dependent information needed by ZMailer programs. This file is created from the Config file in the distribution (the file you are reading right now), and contains variable assignments in an sh-compatible format.

`MAILBIN=`

MAILBIN is the directory hierarchy containing all ZMailer binaries. Usually /usr/lib/mail/bin or /local/lib/mail.

`MAILSHARE=`

MAILSHARE is the directory hierarchy containing site-wide configuration files and databases. Usually /usr/lib/mail or /local/share/mail.

`MAILVAR=`

MAILVAR is the directory that will contain machine-specific configuration files and databases. Usually /usr/lib/mail or /var/db/mail or /local/share/mail.

`MAILBOX=`

MAILBOX is the directory containing all the user mailboxes. This is defaulted inside the mailbox.c program (currently /var/mail) and may be overridden here. Usually /usr/spool/mail or /var/mail.

`POSTOFFICE=`

POSTOFFICE is the directory hierarchy used to manipulate message files, where runtime activity takes places. Usually /usr/spool/postoffice or /var/spool/postoffice.

`ROUTERDIRS=`

Multiple LOWER priorities on message routing can be defined by creating $POSTOFFICE/<component-of-$ROUTERDIRS> -directories. Routers process first $POSTOFFICE/router/ -directory, and once it is empty, files from subsequent dirs. See mail(3) mail_priority These can be only under the $POSTOFFICE.

ROUTERDIRS=router1:router2:router3:router4

`ROUTERDIRHASH=`

When defined, ROUTERDIRHASH submits messages immediately into the 'A' thru 'Z' subdirectory of whatever directory (e.g. see ROUTERDIRS). IF NO SUCH "hash subdirs" EXIST, MESSAGE SUBMISSION WILL FAIL! (The value must be "1" for this to take effect!)

`LOGDIR=`

LOGDIR is the directory where log files will appear. Usually /usr/spool/log or /var/log.

`MANDIR=`

MANDIR is the top of the manual directory hierarchy where manual pages for the ZMailer programs are installed. Usually /usr/man or /local/man.

`LIBRARYDIR=`

LIBRARYDIR is the place for storing libzmailer.a, which can be used to # create programs which use Zmailer's zmailer(3) (aka: mail(3)) -library. LIBRARYDIR= @libdir@

`INCLUDEDIR=`

INCLUDEDIR is the place for storing zmailer.h -- a copy of include/mail.h # and it is used in conjunction with the libzmailer.a .. INCLUDEDIR= @includedir@

`SMTPOPTIONS=`

SMTPOPTIONS are command line options given to the smtpserver when started # from the zmailer shell script. The intent is that if you want non-default # address verification options they can be specified here. The default # value is "-sve". This is also used, when invoking "sendmail" with # "-bs" option. #SMTPOPTIONS= "-l ${LOGDIR}/smtpserver" SMTPOPTIONS= @SMTPOPTIONS@

`ALLOWSOURCEROUTE=`

ALLOWSOURCEROUTE (when present) stops the system from ignoring # the old RFC821/822 source routes of type: @a,@b:c@d; By "ignoring" # we mean here that system chops away "@a,@b:" and uses only: c@d # This is done at all input portals; smtpserver, and at sendmail/rmail. #ALLOWSOURCEROUTE=

`SCHEDULEROPTIONS=`

SCHEDULEROPTIONS are command line options given to the scheduler when # started from the zmailer shell script. The intent is that if you want # non-default logging options, the can be specified here. The default # value is "" # #SCHEDULEROPTIONS= "-l ${LOGDIR}/scheduler.perflog -S -H" SCHEDULEROPTIONS= @SCHEDULEROPTIONS@

`SCHEDULERDIRHASH=`

The SCHEDULERDIRHASH is magic thing to tell to the router that it # should move resulting files directly into hash subdir(s) of the # scheduler subsystem, and not only to the main-level. # Existence of this variable also overrides -H option(s) to # the scheduler. Value is the number of -H options. # If these hash subdirectories don't exist, system failure happens! SCHEDULERDIRHASH=1

`SCHEDULERNOTIFY=`

> # The SCHEDULERNOTIFY defines, where is a socket at which the scheduler # listens for PF_UNIX/SOCK_DGRAM messages telling paths to new jobs. # The router(s) inform the scheduler of new jobs. SCHEDULERNOTIFY=@POSTOFFICE@/.scheduler.notify

`ROUTERNOTIFY=`

> # The ROUTERNOTIFY defines, where is a socket at which the router # listens for PF_UNIX/SOCK_DGRAM messages telling paths to new jobs. # The injection library informs the router queuing subsystem of new jobs. ROUTERNOTIFY=@POSTOFFICE@/.router.notify

`ROUTEROPTIONS=`

> # ROUTEROPTIONS are command line options given to the router when started # from the zmailer shell script. The default values are "-dkn 4" #ROUTEROPTIONS= "-dkn 4" ROUTEROPTIONS= @ROUTEROPTIONS@

`MAILSERVER=`

> # MAILSERVER is the hostname of the remote machine where the postoffice is # located. This value is only needed in an environment with distributed file # systems, and if it exists will be used by the mail queue querying program # as the default name of the host to query. It is a way of overriding the # algorithm used by mailq in an NFS environment, or when you are running a # different kind of DFS. Usually undefined or a hostname. #MAILSERVER= neat.cs

`PUNTHOST=`

> # PUNTHOST is where mail that is supposed to go to a local address, but # no such address exists, is punted to. #PUNTHOST= relay.cs

`FORCEPUNT=`

> # FORCEPUNT is for cases when the local machine under no circumstances # is to store any email locally, but send all such to this given address # (local host is a member on a "cluster" whose message store is at some # other cluster server, and said node handles "local" delivery for all # cluster members... *including* running pipes..) #FORCEPUNT= mailhost

`SMARTHOST=`

> # SMARTHOST is where mail that cannot be resolved or routed is punted to. # There used to be a variable for this, now a better way is to use 'routes' # database at which you put line: . smtp!smart.host.name # (That is: dot, white-space(s), "smtp!smart.host.name")

`NOBODY=`

> # NOBODY is the unprivileged UID value. # This is absolutely necessary if setuid() will fail on your "nobody" account # uid (if it is -2, for example). Make sure that whatever value you give # here will work with setuid(). Values between 1 and 29999 will usually work. # BE CAREFULL WITH THIS! THE SYSTEM RELIES ON IT VERY MUCH IN DEED! # (On SunOS 4.1.x, the value of "-2" works the best, on Solaris the default # for nobody is 60001! If your system has "nobody" "account", use here the # name instead of number -- it should (usually) work) # -- Use a mapping via /etc/passwd, this is most generic.. NOBODY=nobody

`LOGLEVEL=`

> # LOGLEVEL may be set to restrict the log output of the router to entries # whose tags are found in the specified string value. The currently known # tags are: # address: deferred: file:

header_defer: info: recipient: #LOGLEVEL= "file: recipient:" LOGLEVEL= "deferred: file: header_defer:"

`NNTPSERVER=`

Builtin USENET channel uses NNTPSERVER variable (depending upon your # inews ..) to send the artickle to.. NNTPSERVER= @NNTPSERVER@

`INEWSBIN=`

Builtin USENET channel uses NNTPSERVER variable (depending upon your # inews ..) to send the artickle to.. INEWSBIN= @INEWS@

`SENDMAILPATH=`

Where the sendmail (compability one) shall be located ? SENDMAILPATH= @SENDMAILPATH@

`RMAILPATH=`

Where is the rmail to be located at ? RMAILPATH= @RMAILPATH@

`BLOCKLOCKS=`

MAILBOX locking scheme -- no configuration option (yet) # See man-page of mailbox.8 for details; the order of key-chars # is meaningfull: # '.' Dotlock scheme for mailboxes at $MAILBOX/ directory # 'F' flock() locking of files (and perhaps mailboxes) # 'L' lockf() locking of files (and perhaps mailboxes) # ':' Separates the two parts of the parameter; left part # is for the mailbox locking, and right part is for # all other kinds of files. # # We use compiled-in defaults at the mailbox program! # Following examples are for flock(), and lockf() systems with their # respective defaults. (Systems capable to use both will use lockf()) #MBOXLOCKS=".F:F" #MBOXLOCKS=".L:L"

`SELFADDRESSES=`

The SELFADDRESSES is a comma separated list of IP address literals # listing all of our acceptable IP addresses (Comma because IPv6 uses # colon for short-hand notation..): # For usual (IPv4) universe, no addresses are needed listing, however # for IPv6 it may be necessary - likewise if you want to use cluster-mode, # you may want to list all *cluster* addresses here - nodes know only # their local ones, after all.. (See: doc/guides/etrn-cluster) #SELFADDRESSES=[1.2.3.4],[2.3.4.5],[ipv6.::1.2.3.4] SELFADDRESSES=@SELFADDRESSES@

`DBTYPE=`

What kind of DB type we prefer to use ? We can support several, # after all... btree/ndbm/gdbm ... (DBEXT: pag/db/dat) DBTYPE=@DBTYPE@

`DBEXT=`

What kind of DB type we prefer to use ? We can support several, # after all... btree/ndbm/gdbm ... (DBEXT: pag/db/dat) DBEXT=@DBEXT@

`DBEXTtest=`

What kind of DB type we prefer to use ? We can support several, # after all... btree/ndbm/gdbm ... (DBEXT: pag/db/dat) DBEXTtest=@DBEXTtest@

`DEFCHARSET=`

The characterset to be used as a default when turning 8-bit containing # headers to MIME-2 headers -- and what to say at the default generated # "Content-Type: text/plain; charset=XXXX" -header in case the original # message was not of MIME, and still had 8-bit chars... DEFCHARSET=ISO-8859-1

`RFC822TABS=`

We want those nice tabs between the header field name and value # The task of generating TABs or SPACEs is at TA *writeheaders(). # Value '0' here yields expansion of possibly existing header resident # line-start TABs. There is no mechanism to turn line-start SPACEs # to TABs with any other value stored here. RFC822TABS=@RFC822TABS@

`MAX_NR, MAX_NT, MAX_LOAD=`

If the following limitations are exceeded then zmailcheck # will sent an alert. # Limit on the Router Queue MAX_NR=1000 # Limit on Transport Queue MAX_NT=1000 # Load times 100 MAX_LOAD=300

`SYSLOGFLG=`

SYSLOGFLG tells which systems use syslog to log things: # Set of chars which are as follows: # S smtpserver and @SENDMAILPATH@ # R router # T transport agents # C scheduler completion of a message # #SYSLOGFLG=SRT SYSLOGFLG=RT

`TRUSTEDUSER=`

Per default, ZMailer uses “daemon” userid when it wants to # operate in “runastrusteduser()” mode. Finding that userid # (or rather its numeric uid) can be a bit difficult, and if it # *fails*, apparently uid 65535 will be used. # #TRUSTEDUSER=daemon

`ORGDOMAIN=`

Use ORGDOMAIN in ZENV if the system can't generate # MIME multipart boundary string contained host/domain ids # automagically... # #ORGDOMAIN=my.local.domain

`ROUTEUSER_IN_ABNORMAL_UNIX=`

Depending, are you running strange private customer account databases # hooked (only) into 'mailbox', or not, make sure following is non-empty # if you *are* using private databases, as then ZMailer's router won't # claim wronly userid to be nonexistent.. These shunted tests look for # HOMEDIRECTORY, which might be nonexistent thing at such funny systems... # An EMPTY string means "this is NORMAL unix": # # (A "bug" is that this isn't automatically substituted, but non-void # content gives behaviour that has been around for quite a while...) # ROUTEUSER_IN_ABNORMAL_UNIX="@ROUTEUSER_IN_ABNORMAL_UNIX@"

`LISTSERV=`

Some sites (well, one FUNET site), has LISTSERV, this is for # configuring that subpart of the aliases.cf scripts: #LISTSERV=/v/net/listserv.funet.fi

Chapter 7. Verifying the System

FIXME! FIXME! *TO BE WRITTEN !*

Chapter 8. Installing Whoson Service

FIXME! FIXME! *TO BE WRITTEN !*

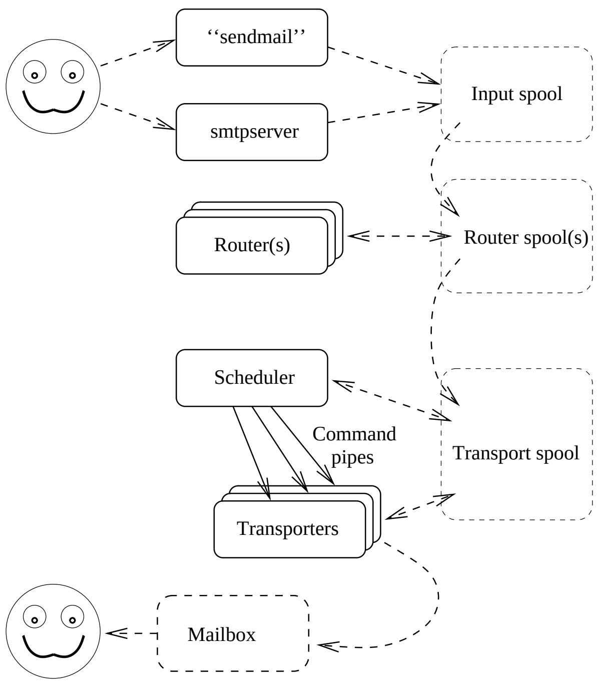
The need for this, and other similar "POP-before-SMTP" approaches has pretty much been obsoleted by development of SMTP AUTHENTICATION, and doing it in particular under TLS wrappers on SUBMISSION port.

III. Administation

This section covers subsystem management issues, including usage and configuration examples, basic and somewhat specific explanations of how pre-existing scripts have been done.

The Figure 14 repeats earlier picture showing central components of the system.

Figure 14. ZMailer's processes.



The things in the picture are pointed further here, along with their related auxiliary programs, etc:

smtpserver

Administration of the **smtpserver** is described at Chapter 12, and detailed Reference is at Chapter 17.

sendmail

The **sendmail** client compatibility functions program is described at Reference Chapter 18.

router

Administration of the **router** is at Chapter 13, and Reference is at Chapter 21.

Auxiliary programs used to support the **router** include:

zmailer newdb

This script has two behaviours, if `$MAILVAR/db/dbases.conf` file can be found, utility script **newdbprocessor** is run. If not, a set of pre-determined individual database regeneration actions is done.

newdb

A perl wrapper for actual internal **makedb** utility taking care of things like correct sequence of file movements after successfull generation of new binary database file(s).

newaliases

Classical `aliases` database regenerator, subset of **zmailer newdb**.

newfqdnaliases

Behaves like the **newaliases**, but processes different database: `fqdnaliases`. Subset of **zmailer newdb**.

scheduler

Administration of the **scheduler** is at Chapter 14, and Reference is at Chapter 22.

Auxiliary programs used to support the **scheduler** include:

mailq

Utility for querying **scheduler**'s view of the universe.

mailbox

Program driven by the **scheduler** to do (usually) delivery to local mailboxes, running pipes, and writing to files.

smtp

Program to do delivery to external systems via SMTP protocol (with many supported extensions).

hold

Handler of "`hold`"-channel deliveries.

sm

There are lots of programs intended to be run under sendmail's `M`-entry (*Mailer-entry*) lines. This program supplies that interface layer to an extent which is meaningfull in ZMailer sense.

errormail

Handler of "`error`"-channel deliveries.

expirer

This is actually driven by the administrator via **manual-expirer** wrapper script.

Chapter 9. DNS and ZMailer

The cornerstone of everything in busy Internet email routing is a properly working DNS server, and a modern resolver library. If you use the BIND nameserver, you should be using a recent version. As of this writing (January 2006), BIND server developers recommend version 9. They also strongly recommend, that you do not let zone data masters (either masters, or slave copies) to do any recursive resolving, and do recursive resolvings with servers that do not have locally mastered data.

You can get improved DNS performance by installing local **named**(8), which does cache replies, including *negative* replies.

For the file `/etc/resolv.conf`:

```
  domain     your.domain
  nameserver 127.0.0.1
  nameserver (some other server)
```

For the local nameserver daemon (named(8)) you should have at least following type of configuration:

```
For 4.* series:  /etc/named.boot
  forwarders 10.12.34.56  10.45.67.89
  options forward-only

For 8.* series:  /etc/named.conf
  options {
      forward only;
      forwarders {
             10.12.34.56;
             10.45.67.89;
      };
  };
```

which means that all the queries are attempted to be resolved by the servers at IP addresses *10.12.34.56* and *10.45.67.89*, and both the local server, and remote servers will cache DNS responses.

For Solaris, Linux, and some other environments you propably have file `/etc/nsswitch.conf`. There the interesting line is one referring with "*hosts:*" tag. In most cases the default setup assumes you will use e.g. NIS(+) in the system overriding DNS and/or local files. In general that is quite *bad* thing to do — especially for DNS intensive application, like mailers... Suggested value:

```
  hosts: files dns
```

At DEC Tru64 there is another file with same purpose as `nsswitch.conf`, it is: `/etc/svc.conf`.

At Solaris 2.6 (and after?) there is also a "**nscd**" daemon (name service cache daemon), which has appeared at times to *harm* DNS lookup intensive systems. At its configuration file `/etc/nscd.conf`:

```
     enable-cache    hosts   no
```

Same trouble entity appears also at Linuxes with glibc 2.*. That thing appears (at both systems) to turn temporary lookup failures to permanent kind of errors. E.g. confuse return codes!

Chapter 10. Security Issues

The intention of the security mechanisms is not to prevent address faking, but to control the privileges which are used to execute pipes, and accessing files (read, and write).

In addition of doing strict privilege control on who can do what, ZMailer has a concept of *trust*, which shows as a group of accounts who can claim wildly “fradulent” data in their headers.

The *trusted* accounts are those listed in the ZMailer group or the “trusted” variable in the system configuration (`router.cf`) file.

Of course when one uses SMTP protocol to inject email, it is extremely easy to “fake” any source and destination envelope and visible addresses.

Having local-parts that allow delivery to arbitrary files, or which can trigger execution of arbitrary programs, can clearly lead to a huge security problem. **sendmail** does address this problem, but in a restrictive, and unintuitive manner. This aspect of ZMailer security has been designed to allow the privileges expected by common sense.

The responsibility for implementing this kind of security is split between the **router**, and the *transport agent* that delivers a message to an address. Since it is the Transport Agent that must enforce the security, it needs some information to guide it. Specifically, for each address it delivers to, some information about the “*trustworthiness*” of that address is necessary so that the transport agent can determine which privileges it can assume when delivering for that destination. This information is determined by the **router**, and passed to the transport agent in the message control file. The specific measure of trustworthiness chosen by ZMailer, is simply a numeric user id (uid) value *representing the source of the address.*

When a message comes into the mailer from an external source, the destination addresses should obviously have no privileges on the local host (when mailing to a file or a program). Similarly, common sense would indicate that locally originated mail should have the same privileges as the originator. Based on an initial user id assigned from such considerations, the privilege attached to each address is modified by the attributes of the various alias files that contain expansions of it. The algorithm to determine the appropriate privilege is to use the user id of the owner of the alias file if and only if that file is not group or world writable, and the directory containing the file is owned by the same user and is likewise neither group nor world writable. If any of these conditions do not hold, an unprivileged user id will be assigned as the privilege level of the address.

It is entirely up to the transport agent whether it will honour the privilege assignment of an address, and indeed in many cases it might not make sense (for example for outbound mail). However, it is strongly recommended that appropriate measures are taken when a transport agent has no control over some action that may affect local files, security, or resources.

The described algorithm is far from perfect. The obvious dangers are:

- The grandparent directories, to the Nth degree, are ignored, and may not be secure. In that case all security is lost.
- There is a window of vulnerability between when the permissions are checked, and the delivery is actually made. This is the best argument for embedding the entire local-aliasing into the local-delivery agent program.

There is also another kind of security that must be addressed. That is the mechanism by which the **router** is told about the origin of a message. This is something that must be possible for the message receiving programs (**/bin/rmail** and the *SMTP server* are examples of these) to specify to ZMailer.

The **router** knows of a list of trusted accounts on the system. If a message file is owned by one of these user id’s, any sender specification within the message file will be believed by ZMailer. If the message file is not owned by such a trusted account, the **router** will cross-check the message file owner with any stated “From:” or “Sender:” address in the message header, or any origin specified in the envelope. If a discrepancy is discovered, appropriate action will be taken. This means that there is no way to forge the internal origin of a message without access to a trusted account.

Chapter 11. The Queue

Normal processing within ZMailer goes via directories described at Figure 11-1. A message *may* get sidelined or otherwise linked into other directories for several possible reasons.

Important: *A filesystem without following three properties is not suitable for ZMailer's* `$POSTOFFICE/`:

- *Files must succeed to be `link`(2):ed in between directories within the filesystem without copying them.*
- *The files must have i-node numbers available via `fstat`(2) calls, and those numbers must be uniquely representable with 31 bits wide integers.*
- *The file i-node numbers must not change with `rename`(2) or `link`(2) calls applied to them.*

Most of the `$POSTOFFICE/` directory must be a single mounted filesystem within which files can be `link`(2)ed from one directory to another, as well as moved around with `rename`(2)

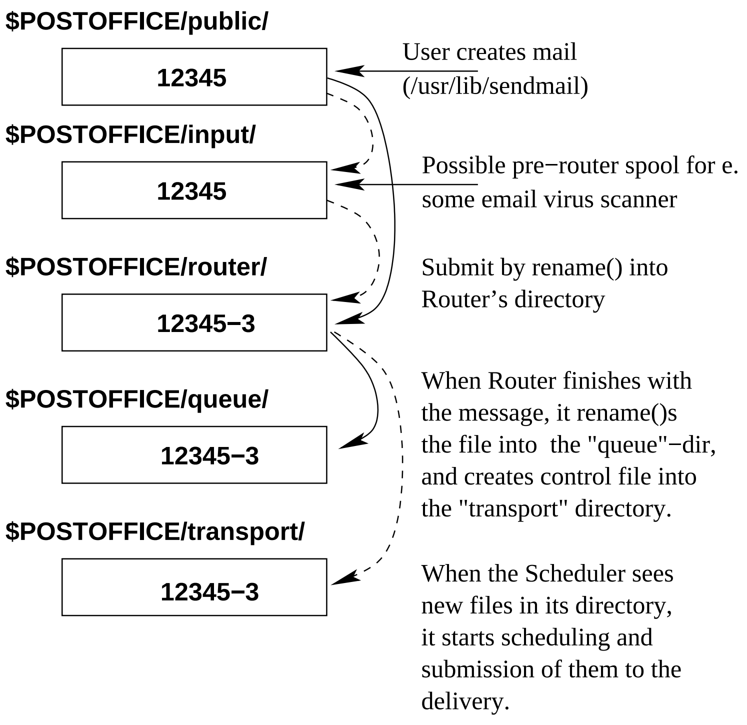
However, the `$POSTOFFICE/transport/` subdirectory *can be* separate filesystem mounted under `$POSTOFFICE/`! Such arrangement can (under some situations) result in additional system performance, as transport agents need to modify (write locks) files in the `$POSTOFFICE/transport/` subdirectory, while they only read (without locks) files in `$POSTOFFICE/queue/` subdirectory.

Adding system reliability in form of having directory data committed to disk at the time of the directory modifying operations returning with success is a nice bonus, although in normal UFS-like cases that taxes system performance heavily.

E.g. running fast-and-loose with async metadata updates in Linux EXT2 filesystem gives you performance, but in case the system crashes, your postoffice directory may be in shambles, and important email may have been lost.

How exactly you can combat the problem is yours to choose. Most filesystems for UNIX have lots of different options at mount-time, and also by-directory attributes can be set to control these things. Check yours after your decide on what kind of data loss threat you can tolerate at the expence of what speed reduction. (E.g. 300+ day straight uptime with power surges during a thunderstorm at the end of it toasting your machine along with its disks and filesystems, but trouble-free running until then ?)

Figure 11-1. Directories that ZMailer uses for message processing.


$POSTOFFICE/public/
12345
User creates mail (/usr/lib/sendmail)
$POSTOFFICE/input/
12345
Possible pre−router spool for e. some email virus scanner
$POSTOFFICE/router/
12345−3
Submit by rename() into Router's directory
$POSTOFFICE/queue/
12345−3
When Router finishes with the message, it rename()s the file into the "queue"−dir, and creates control file into the "transport" directory.
$POSTOFFICE/transport/
12345−3
When the Scheduler sees new files in its directory, it starts scheduling and submission of them to the delivery.

There are multiple queues in ZMailer. Messages exist in in one of five locations:

- Submission temporary directory (`$POSTOFFICE/public/`)
- Freezer directory (`$POSTOFFICE/freezer/`)
- Router directory (`$POSTOFFICE/router/`)
- Deferred directory (`$POSTOFFICE/deferred/`)
- Scheduler directories (`$POSTOFFICE/transport/`, `$POSTOFFICE/queue/`)

And sometimes is also copied into the:

- Postmaster analysis area (`$POSTOFFICE/postman/`)

11.1. Message Submission Areas

In few places inside of ZMailer (in parts of router, and more so in parts of scheduler) the system expects the filenames to be decimal encodings of integers of 31 bits (maybe 63 bits at systems with suitably large 'long'), and those integers (modulo something) are used as keys in several internal database lookups.

The numeric values used in filenames must be unique for the entire lifetime of the spool files.

Message submission is done by writing a temporary file into the directory (`$POSTOFFICE/public/`), the actual format of the submitted message is described in Appendix E.

When the temporary file is completely written, flushed to disk, and closed, it is then renamed into one of the **router** input directories, usually into `$POSTOFFICE/router/` with a name that is a decimal representation of the spool-file i-node number. This is a way to ensure that the name of the file in the `$POSTOFFICE/router/` directory is unique.

The message may also be renamed into alternate router directories, which give lower priorities on which messages to process when.

Sometimes, especially **smtpserver** built files may be moved into alternate directories. The **smtpserver** "ETRN" command has two implementations, original one is by moving the built special file to the directory `$POSTOFFICE/transport/` without going through the **router**. The **smtpserver** may also move newly arrived files into the `$POSTOFFICE/freezer/` directory.

11.2. Router Behaviour on Queues

```
FIXME:
   This description is from era before the router got
   "daemonized" in a sense of having separate instance
   of queue processor (and it also handles logging/log
   rotation) and a worker-farm of routing work processes.
```

FIXME: The system can have multiple **router** processes running in parallel and competeting on input files. Multiple processes may make sense when there are multiple processors in the system allowing running them in parallel, but also perhaps for handling cases where one process is handling routing of some large list, and other (hopefully) will get less costly jobs.

The **router** processes have a few different behaviours when they go over their input directories.

First of all, if there are `ROUTERDIRS` entries, those are scanned for processing after the primary `$POSTOFFICE/router/` directory is found empty.

Within each directory, the **router** will sort files at first into mod-time order, and then process the oldest message first. (Unless the **router** has been started with the "`-s`" option.)

The **router** acquires a lock on the message (spool file) by means of renaming the file from its previous name to a name created with formatting statement:

```
sprintf(buf, "%ld-%d", (long)filestat.st_ino, (int)getpid());
```

Once the **router** has been able to acquire a new name for the file, it starts off by creating a temporary file of **router** routing decisions. The file has a name created with formatting statement:

```
sprintf(buf, "..%ld-%d", (long)filestat.st_ino, (int)getpid());
```

Once the processing has completed successfully, the original input file is moved into the directory `$POSTOFFICE/queue/`, and the **router** produced **scheduler** work-specification file is moved to the `$POSTOFFICE/transports/` directory with the *same name* that the original file has.

If the routing analysis encountered problems, the message may end up moved into the directory `$POSTOFFICE/deferred/`, from which the command **zmailer resubmit** is needed to return the messages into processing (The **router** logs should be consulted for the reason why the message ended up in the `/deferred/` area, especially if the command **zmailer resubmit** is not able to get the messages processed successfully and the files end up back in the `/deferred/` area.)

If the original message had errors in its RFC-822 compliance, or some other detail, a copy of the message may end up in the directory `$POSTOFFICE/postman/`.

See Postmaster Analysis Area on section Section 11.4.

11.3. Scheduler, and Transport Agents

The scheduler work specification files are in the directory `$POSTOFFICE/transport/`, under which there can be (optionally) one or two levels of subdirectories into which the actual work files will be scattered to lessen the sizes of individual directories in which files reside, and thus to speed up the system implied directory lookups at the transport agents, when they open files, (and also in the scheduler).

When the **router** has completed message file processing, it places the resulting files into the top level directory of the **scheduler**; `$POSTOFFICE/transport/`, and `$POSTOFFICE/queue/`.

The **scheduler** (if so configured by "`-H`" option) will move the messages into "hashed subdirectories," when it finds the new work specification files, and then start processing them.

The transport agents are run with their CWD in directory `$POSTOFFICE/transport/`, and they open files relative to that location. Actual message bodies, when needed, are opened with the path prefix "`../queue/`" to the work specification file name.

Usually it is the *transport agent's* duty to log different permanent status reports (failures, successes) into the end of the work-specification file. Sometimes the **scheduler** also logs something at the end of this file. All such operations are attempted *without* any sort of explicit locking, instead trusting the *write(2)* system call to behave in an atomic manner while writing to the disk file, and having a single buffer of data to write.

Once the **scheduler** has had all message recipient addresses processed by the *transport agents*, it will handle possible diagnostics on the message, and finally it will remove the original spool-file from the `$POSTOFFICE/queue/`, and the work-specification file from `$POSTOFFICE/transport/`.

11.4. Postmaster Analysis Area

If the filename in the `$POSTOFFICE/postman/` directory has an underscore in it, the reason for the copy is *soft*, that is, the message has been sent through successfully in spite of being copied into the directory.

If the filename in that directory does not have an underscrore in it, that file has not been processed successfully, and the only copy of the message is now in that directory!

Usually forementioned underscoreless filenames are double-errors, that is, error messages to error messages. There is nowhere else to send them.

The indication of *error* message is, of course, `MAIL FROM:<>` per RFC 821.

If the **smtpserver** receives a message with content that the *policy filtering* system decides to be dubious, it can move the message into `$POSTOFFICE/freezer/` directory with a bit explanatory name of type:

```
sprintf(buf, "%ld.%s", (long)filestat.st_ino, causecodestring);
```

The files in the freezer-area are in the input format to the **router**, and as of this writing, there are no tools to automatically process them for obvious spams, and leave just those that were falsely triggered.

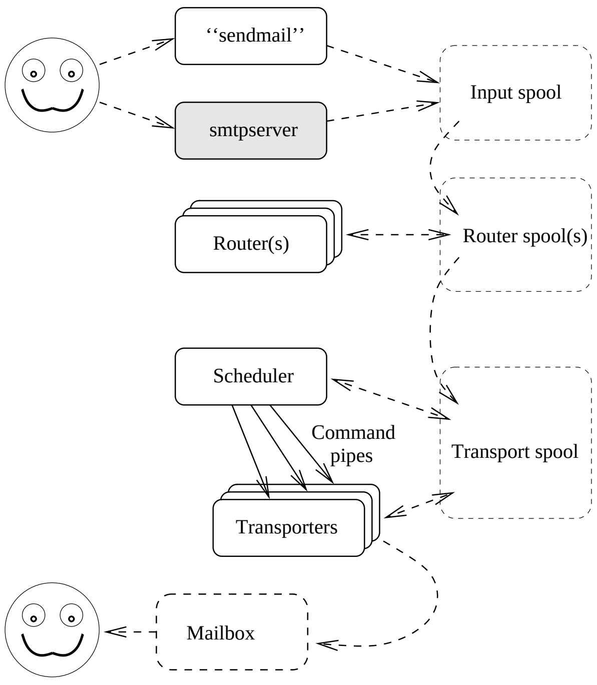
Chapter 12. Smtpserver Administration

```
Things to place here
 - administrative stuff
 - runtime command-line parameters (most important of them)
 - smtpserver.conf
   - PARAM entries (most common/important ones)
   - SMTP policy-control
   - content-policy interface
```

The **smtpserver** is ZMailer's component to receive incoming email via SMTP protocol. Be it thru TCP channel, or thru Batch-SMTP.

The Figure 12-1 repeats earlier picture showing central components of the system, and where the **smtpserver** is in relation to them all.

Figure 12-1. ZMailer's processes; Smtpserver



The **smtpserver** program actually has several operational modes.

- It can operate as a stand-alone internet service socket listener, which forks off childs that do the actual SMTP-protocol service.
- It can be started from under the control of the **inetd**(8) server, and it can there fulfill most of the the same roles as it does in the stand-alone mode.
- It can even be used to accept Batch-SMTP from incoming files (UUCP, and BITNET uses, for example).

The runtime command-line options are as follows:

smtpserver [-46aignvBV] [-p *port*] [-l *logfile*] [-s 'strict'] [-s *[ftveR]*] [-L *maxloadaver*] [-M *SMTPmaxsize*] [-P *postoffice*] [-R *router*] [-C *cfgfile*] [-T *IPv4/IPv6-address-literal*]

The most commonly used command line options are:

smtpserver [-aBivn] [-s *ehlo-styles*] [-l *logfile*] [-C *cfgpath*]

Without any arguments the **smtpserver** will start as a daemon listening on TCP port 25 (SMTP).

Most important of the options are:

`-a`

Query IDENT information about the incoming connection. This information (if available at all) may, or may not tell who is forming a connection.

`-B`

The session is Batch-SMTP a.k.a. BSMTP type of session. Use of "`-i`" option is needed, when feeding the input batch file.

`-i`

This is interactive session. I/O is done thru *stdin/stdout*.

`-v`

Verbose trace written to stdout for use in conjunktion with "`-i`", and "`-B`".

`-n`

This tells that the **smtpserver** is running under **inetd**(8), and that the stdin/stdout file handles are actually network sockets on which we can do peer identity lookups, for example.

`-s` *`ehlo-style`*

Default value for various checks done at SMTP protocol MAIL FROM, RCPT TO, VRFY, and EXPN commands. These are overridden with the value from EHLO-patterns, if they are available (more below)

`-s 'strict'`

Special value directing the system to be extremely picky about the incoming SMTP protocol — mainly for protocol compliance testing, usually way too picky for average (sloppy) applications out there...

`-l` *`logfilepath`*

Filename for the **smtpserver** input protocol log. This logs about everything, except actual message data...

`-l 'SYSLOG'`

This tells **smtpserver** that it shall send all incoming smtp protocol transactions via `syslog` facility to elsewere. Used syslog parameters are: FACILITY=mail, LEVEL=debug.

This option may be used in addition to the preceding file logging variant. Double-use of file referring variant uses the last defined file, but this doesn't affect files at all.

`-C` *`configfilepath`*

Full path for the **smtpserver** configuration in case the default location can not be used for some reason.

`-M` *`SMTPmaxsize`*

SMTPmaxsize defines the absolute maximum size we accept for incoming email. (Default: infinite) (This is a local policy issue.)

`-T` *`[IPv4-or-IPv6-address-literal]`*

An *address literal* is used in interactive test mode to check how the rules work with given inputs when the source address of the connection is what is given in headers.

12.1. `smtpserver.conf`

If the system has file `$MAILSHARE/smtpserver.conf` (by default), that file is read for various parameters, which can override most of those possibly issued at the command line.

Example configuration is is in figure Figure 12-2.

Figure 12-2. Sample `smtpserver.conf` file

```
#PARAM maxsize          10000000    # Same as -M -option
#PARAM max-error-recipients    3    # More than this is propably SPAM!
#PARAM MaxSameIpSource        10    # Max simultaneous connections from
#                                   # any IP source address
#PARAM ListenQueueSize        10    # listen(2) parameter
#
# Enables of some commands:
#PARAM  DEBUGcmd
#PARAM  EXPNcmd
#PARAM  VRFYcmd
#PARAM  enable-router # This is a security decission for you.
#                   # This is needed for EXPN/VRFY and interactive
#                   # processing of MAIL FROM and RCPT TO addresses.
#                   # However it also may allow external user entrance
#                   # to ZMailer router shell environment with suitably
#                   # pervert input, if quotation rules are broken in
#                   # the scripts.
PARAM help -------------------------------------------------------------
PARAM help  This mail-server is at Yoyodyne Propulsion Inc.
PARAM help  Our telephone number is: +1-234-567-8900, and
....
PARAM help -------------------------------------------------------------

# The policy database:  (NOTE: See  'makedb'  for its default suffixes!)
```

```
PARAM  policydb  $DBTYPE  $MAILVAR/db/smtp-policy

# External program for received message content analysis:
#PARAM  contentfilter   $MAILBIN/smtp-contentfilter
....
#
# TLSv1/SSLv[23] parameters; all must be used for the system to work!
#
# See  doc/guides/openssl,  or:
# http://www.aet.tu-cottbus.de/personen/jaenicke/pfixtls/doc/setup.html
#
#PARAM  use-tls
#PARAM  tls-CAfile      $MAILVAR/db/smtpserver-CAcert.pem
#PARAM  tls-cert-file   $MAILVAR/db/smtpserver-cert.pem
#PARAM  tls-key-file    $MAILVAR/db/smtpserver-key.pe
....
#
# HELO/EHLO-pattern     style-flags
#               [max loadavg]
#
localhost           999 ftveR
some.host.domain    999 !NO EMAIL ACCEPTED FROM YOUR MACHINE

# If the host presents itself as:  HELO [1.2.3.4], be lenient to it..
# The syntax below is due to these patterns being SH-GLOB patterns,
# where brackets are special characters.

\[*\]               999 ve

# Per default demant strict syntactic adherence, including fully
# qualified addresses for  MAIL FROM, and RCPT TO.  To be lenient
# on that detail, remove the "R" from "veR" string below:

*                   999 veR
```

12.1.1. `smtpserver.conf`; PARAM keywords

The PARAM keywords and values are:

`maxsize` *nn*

Maximum size in the number of bytes of the entire spool message containing both the transport envelope, and the actual message. That is, if the max-size is too low, and there are a lot of addresses, the message may entirely become undeliverable..

`max-error-recipients` *nn*

In case the message envelope is an error envelope (MAIL FROM:<>), the don't accept more than this many separate recipient addresses for it. The default value is 3, which should be enough for most cases. (Some SPAMs claim to be error messages, and then provide a huge number of recipient addresses...)

`MaxSameIpSource` *nn*

(Effective only on daemon-mode server — not on "`-i`", nor "`-n`" modes.) Sometimes some systems set up multiple parallel connections to same host (qmail ones especially, not that ZMailer has entirely clean papers on this - at least up to 2.99.X series), we won't accept more than this many connections from the same IP source address open in parallel. The default value for this limit is 10.

`ListenQueueSize` *nn*

This relates to newer systems where the `listen`(2) system call can define higher limits, than the traditional/original 5. This limit tells how many nascent TCP streams we can have in SYN_RCVD state before we stop answering to incoming SYN packets requesting opening of a connection.

There are entirely deliberate denial-of-service attacks based on flooding to some server many SYNS on which it can't send replies back (because the target machines don't have network connectivity, for example), and thus filling the back-queue of nascent sockets. This can also happen accidentally, as the connectivity in between the client host, and the server host may have a black hole into which the SYN-ACK packets disappear, and the client thus will not be able to get the TCP startup three-way handshake completed.

Most modern systems can have this value upped to thousands to improve systems resiliency against malicious attacks, and most likely to provide complete immunity against the accidental "attack" by the failing network routing.

`DEBUGcmd`

FIXME! WRITEME! #PARAM DEBUGcmd

`EXPNcmd`

FIXME! WRITEME! #PARAM EXPNcmd

`VRFYcmd`

FIXME! WRITEME! #PARAM VRFYcmd

`enable-router`

FIXME! WRITEME! #PARAM enable-router # This is a security decission for you.

`help` *'string'*

This one adds yet another string (no quotes are used) into those that are presented to the client when it asks for "HELP" in the SMTP session.

`PolicyDB` *dbtype dbpath*

This defines the database type, and file path prefix to the binary database containing policy processing information. More of this below. Actual binary database file names are formed by appending type specific suffixes to the path prefix. For example NDBM database appends ".pag" and ".dir", while BSD-Btree appends only ".db". (And the latter has only one file, while the first has two.)

For an operative overview, see Section 12.2, and for deeper details, see Section 17.4.

`contentfilter` *programpath*

The *contentfilter* studies the received message at the end of the DATA or BDAT transaction, and produces syncronous report about should be message be accepted or not. Unlike the `PolicyDB`, this does not (should not) care about validity of envelope source and recipient address validities, although perhaps it should consider at least the recipients in some cases -- e.g. accept about anything when the destination is `<postmaster>`.

For an operative overview, see Section 12.3, and for deeper details, see Section 17.5.

12.1.2. `smtpserver.conf`; "EHLO-style options"

All lines that are not comments, nor start with uppercase keyword "POLICY" are "EHLO-style patterns". This is the oldest form of configuring the **smtpserver**, and as such, it can be seen...

Behaviour is based on glob patterns matching the *HELO/EHLO* name given by a remote client. Lines beginning with a "#", or whitespace are ignored in the file, and all other lines must consist of two tokens: a shell-style (glob) pattern starting at the beginning of the line, whitespace, and a sequence of style flags. The first matching line is used. As a special case, the flags section may start with a ! character in which case the remainder of the line is a failure comment message to print at the client. This configuration capability is intended as a way to control misbehaving client software or mailers.

The meanings of the style flag characters are as follow:

`f`

Check "*MAIL FROM*" addresses through online processing at the attached **router** process

`t`

Check *RCPT TO* addresses through online processing at the attached **router** process

`v`

Allow execution of *VRFY* command online at the attached **router** process

`e`

Allow execution *EXPN* command online at the attached **router** process

`R`

Require addresses to be in fully qualified (domained) form: "*local@remote*" (strict 821)

`S`

Allow Sloppy input for systems incapable to respect RFC 821 properly; WinCE1.0 does: "*MAIL FROM:user@domain*" :-(

12.2. Policy Based Relaying Control

The policy database that **smtpserver** uses is built with **policy-builder.sh** script, which bundles together a set of policy source files:

File	What

File	What
`DB/smtp-policy.src`	*The boilerplate*
`DB/localnames`	("= _localnames")
`DB/smtp-policy.relay.manual`	("= _full_rights")
`DB/smtp-policy.relay`	("= _full_rights")
`DB/smtp-policy.mx.manual`	("= _relaytarget")
`DB/smtp-policy.mx`	("= _relaytarget")
`DB/smtp-policy.spam.manual`	("= _bulk_mail")
`DB/smtp-policy.spam`	("= _bulk_mail")

*If you want, you can modify your `smtp-policy.src` boilerplate file as well as your installed **policy-builder.sh** script.*

Basically these various source files (if they exist) are used to combine knowledge of valid users around us:

`smtp-policy.src`

The controlling boilerplate, *which you should modify!*

`localnames`

Who we are — ok domains for receiving.

`smtp-policy.relay`

Who can use us as outbound relay.

Use `[ip.number]/maskwidth` here for listing those senders (networks) we want to trust. You may also use domains, or domain suffixes so that the IP-reversed hostnames are accepted (but that is a it risky thing due to ease of fakeing the reversed domain names):

```
[11.22.33.00]/24
ip-reversed.host.name
.domain.suffix
```

Server sets its internal "`always_accept`" flag at the source IP address tests before it decides on what to tell to the contacting client. The flag is not modified afterwards during the session.

Usage of domain names here is discouraged as there is no way to tell that domain "`foo.bar`" here has different meaning than same domain elsewere — at "`smtp-policy.mx`," for example.

`smtp-policy.mx.manual`
`smtp-policy.mx`

Who really are our MX clients. Use this when you really know them, and don't want just to trust that if recipient has MX to you, it would be ok. . .

You can substitute this knowledge with a fuzzy feeling by using "`acceptifmx -`" attribute at the generic boilerplate. List here domain names, possibly suffixes:

```
mx-target.dom
.mx-target.dom
```

The suffix ("`.mx-target.dom`") *does not* match to the "dot-less" domain name: "`mx-target.com`" !

You CAN also list here all POSTMASTER addresses you accept email routed to:

```
postmaster@local.domain
postmaster@client.domain
```

these are magic addresses that email is accepted to, even when everything else is blocked.

`smtp-policy.spam.manual`
`smtp-policy.spam`

Those users, and domains that are absolutely no-no for senders, or recipients no matter what earlier analysis has shown. (Except for those senders that we absolutely trust..)

```
user@domain
user@
domain
```

At the “`smtp-policy.src`” boiler-plate file there is one particular section containing default setting statements. See figure Figure 12-3 for the salient details concerning this.

Figure 12-3. The `smtp-policy.src` file default settings fragment

```
...
#| ===========================
#|
#| Default handling boilerplates:
#|
#|   "We are not relaying between off-site hosts, except when ..."
#|
#| You MUST uncomment one of these default-defining pairs, or the blocking
#| of relay hijack will not work at all !
#|
#| == 1st alternate: No MX target usage, no DNS existence verify
#|    Will accept for reception only those domains explicitely listed
#|    in “smtp-policy.mx” and  “localnames”  files.  Will not do
#|    verifications on validity/invalidity of source domains:&lt;foo@bar&gt;
#
# .             relaycustomer - relaytarget -
# [0.0.0.0]/0   relaycustomer - relaytarget -
#
#| == 2nd alternate: No MX target usage, DNS existence verify
#|    Like the 1st alternate, except will verify the sender
#|    (MAIL FROM:&lt;..&gt;) address for existence of the DNS MX and/or
#|    A/AAAA data -- e.g. validity.
#|    If RBL parameters are set below, will use them.
#
# .             relaycustomer - relaytarget - senderokwithdns + = _rbl1
# [0.0.0.0]/0   relaycustomer - relaytarget - senderokwithdns + = _rbl0
#
#| == 3rd alternate: MX relay trust, DNS existence verify
#|    For the people who are in deep s*...  That is, those who for some
#|    reason have given open permissions for people to use their server
#|    as MX backup for their clients, but don’t know all domains valid
#|    to go thru...  Substitutes accurate data to user’s whimsical DNS
#|    maintenance activities.  Vulnerable to inbound MX resource abuse.
#|    If RBL parameters are set below, will use them.

.               relaycustomer - acceptifmx - senderokwithdns + = _rbl1
[0.0.0.0]/0     relaycustomer - acceptifmx - senderokwithdns + = _rbl0

#| == 4th alternate: Sender & recipient DNS existence verify
#|    This is more of an example for the symmetry’s sake, verifies that
#|    the source and destination domains are DNS resolvable, but does not
#|    block relaying
#
# .             senderokwithdns - acceptifdns -
# [0.0.0.0]/0   senderokwithdns - acceptifdns -
#
#|
#|  You may also add  “test-dns-rbl +”  attribute pair to [0.0.0.0]/0
#|  of your choice to use Paul Vixie’s  http://maps.vix.com/  MAPS RBL
#|  system.
#|
#| These rules mean that locally accepted hostnames MUST be listed in
#| the database with “relaytarget +” attribute.
#|
#| ==========================
#|
#| RBL type test rules:

#| First RBL variant: NONE OF THE RBL TESTS
_rbl0 # Nothing at early phase
_rbl1 # Nothing at late phase

#| Second RBL variant: Early block with RBL+DUL+RSS
#_rbl0 test-dns-rbl      +:dul.maps.vix.com:relays.mail-abuse.org
#_rbl1 # Nothing at late phase

#| Third RBL variant: Late block with RBL+DUL+RSS
#_rbl0 rcpt-dns-rbl      +:dul.maps.vix.com:relays.mail-abuse.org
#_rbl1 test-rcpt-dns-rbl +

#|  (The "+" at the DNS zone defines is treated as shorthand to
#|   "rbl.maps.vix.com")
#|
#|  The Third RBL variant means that all target domains can all by
#|  themselves choose if they use RBL to do source filtering.
#|  The “= _RBL1” test *must* be added to all domain instances
#|  where the check is wanted.
#|  (Including the last-resort domain default of ".")
#|  (Or inverting: If some recipient domain is *not* wanting RBL-type
#|   tests, that domain shall have  “test-rcpt-dns-rbl -” attribute
#|   pair given for it at the input datasets - consider smtp-policy.mx
#|   file!)
#|
#| These rules mean that locally accepted hostnames MUST be listed in
#| the database with  ’relaytarget +’ attribute.  ("acceptifmx *" allows
#| reception if the local system is amonst the MXes.)
#|
#| ==========================
#|
#| If your system has “whoson” server (see contrib/whoson-*.tgz),
#| you can activate it by adding  ’trust-whoson +’ attribute pair to
#| the wild-card IP address test:  [0.0.0.0]/0  of your choise.
#|
#| ==========================
#|
#| For outbound relaying control for fixed IP address networks, see
#| comments in file:  smtp-policy.relay
#|
#| ==========================
...
```

12.3. Content Based Filtering

The ZMailer *can* do also message content analysis with an external program at the end of DATA-dot phase, and BDAT LAST phase (that is, when the input message is complete, and final acknowledgement is expected by the email sender.)

The program becomes active if PARAM entry “contentfilter” is set:

```
# External program for received message content analysis:
#PARAM  contentfilter  $MAILBIN/smtp-contentfilter
```

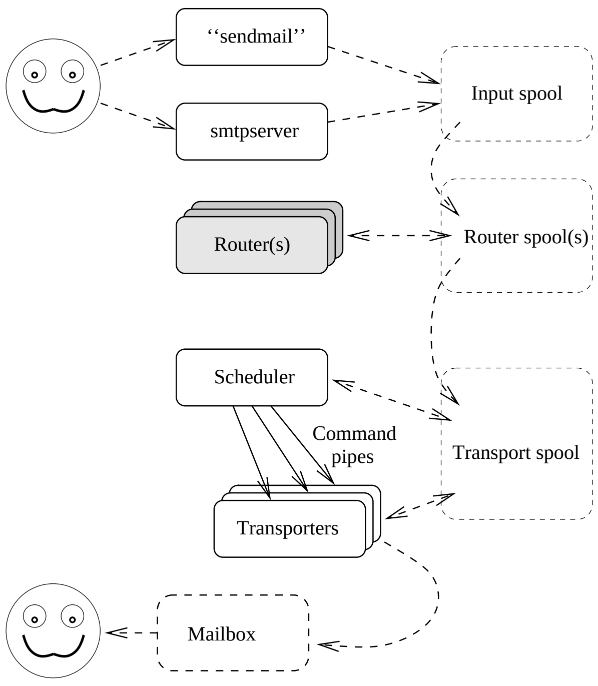
More details are at the Reference part: Section 17.5.

Chapter 13. Router Administration

The **router** is the part of the ZMailer that uses algorithms, and control databases to determine what latter stages, like **scheduler** should do to the message.

The Figure 13-1 repeats earlier picture showing central components of the system, and where the **router** is in relation to to all.

Figure 13-1. ZMailer's processes; Router



In following our intention is to cover topics of:

- What input data router uses
- What output it produces
- How the router is configured, including of what is 'dbases.conf' file.
- How it can be tuned

```
FIXME:
  - Intro to what the router does to the message
  - How the configuration scripts are loaded
  - How the standard scripts are tunable by means of databases,
    specifically 'dbases.conf'
  - The ROUGHT logic of the standard scripts
  - What to do if one wants to tune ?
```

The names (determined at compile-time) and interface specifications for the routing and crossbar functions, are the only crucial "magical" things one needs to contend with in a proper **router** configuration. The syntax and semantics of the configuration file's contents are dealt with in the following subsection. The details of the two functions introduced here are specified after that, once the necessary background information has been given.

Router behavior is controlled by a configuration file read at startup. It is a **zmsh**(1) script that uses facilities provided built into the **router**.

The configuration file looks like a *Bourne Shell* script at first glance. There are minor syntax changes from standard **sh**(1), but the aim is to be as close to the *Bourne Shell language* as is practical. In fact some aspects of variable handling are more of PERL style, and others are even LISPish.

The contents of the file are compiled into bytecode, which can then be interpreted by the **router**. The configuration file is usually self-contained, although an easy mechanism exists to make use of external UNIX programs when so desired. Together with a very flexible database lookup mechanism, functions, and address manipulation based on token-matching regular expressions, the configuration file language is an extremely flexible substrate to accomplish its purpose. When the language is inadequate, or if speed becomes an issue, it is possible to call built in (C coded) functions. The interface to these functions is mostly identical to what a standalone program would expect (modulo symbol name clashes and return values), to ease migration of external programs to inclusion in the **router** process.

13.1. Configuration File Programming Language

Whenever the **router** process starts, its first action is to read its configuration file. The configuration file is a text file which contains statements interpreted immediately when the file is read. Some statements are functions, in which case the function is defined at that point in reading the configuration file. The purpose of the configuration file is to provide a simple way to customize the behavior of routing process of the mailer, and this is primarily achieved by defining the `router` (at Section 21.4.2), and `crossbar` (at Section 21.4.3) functions. For these to work properly, some initialization code and auxiliary functions will usually be needed.

At first sight, a configuration file looks like a Bourne shell script. The ideal is to duplicate the functionality, syntax, and to a large degree the semantics, of a shell script. Therefore, the configuration file programming language is defined in terms of its deviation from standard Bourne shell syntax and semantics. The present differences are:

- No `repeat` statement.
- Functions are allowed, parameter lists are allowed. If not enough arguments are present in a function call to exhaust the parameter list, the so-far unbound parameter variables are bound to `""` (the empty string) as local variables. For example, this is the identity address rewriting function:

  ```
  null (address) {
    return $address  # surprise!
  }
  ```

- Multiple-value returns are allowed. The `return` statement can be used to return a non-"" (non-empty string) value from a function. The following are all legal `return` statements:

  ```
  return
  return $address
  return $channel ${next_host} ${next_address}
  ```

- Variables are dynamically scoped, local variables are the ones in a function's parameter list and those declared with the "`local`" statement. Only the first value of a multiple-value return may be assigned to a variable. All values are either strings, or lists, so no type information, checking, or declaration, is necessary.
- Quoting is a bit stilted. All quotes (double-, single-, back-), must appear in matching pairs at the beginning and at the end of a word.

  {\bf\large CHECK!} Single quotes are not stripped, double quotes cause the enclosed character sequence to be collected into a quoted-string RFC822 token.

  For example, the statement:

  ```
  foo ‘bar "‘baz‘"‘
  ```
  is evaluated as

  ```
  (apply 'foo (apply 'bar (baz)))
  ```
- In standard shells the IFS guides on how variable expansion results are to be treated. Namely in cases where the expansion happens without being enclosed into double-quotes, the expansion result is at first split with IFS contained characters forming the separation sequences.

  ZMailer's "shell" behaves alike PERL in this regard, and *will not do IFS interpolation on the result.* However, unlike with PERL, double-quoted evaluation will not have its contents re-evaluated.

  Thus it is equally safe to do assignments like:

  ```
  var1='some text here'
  var2=' more text
  cat1="$var1$var2"
  cat2=$var1$var2
  ```

  The notable thing at this particular example is that *both* result variables are catenates of the input strings.

  However! If either of inputs is a list of any kind, then the catenate is not to be done this way! See `lappend`.
- Due to lack of implicite split by the IFS characters, ZMailer "shell" contains function `ifssplit`.
- The `for` construct is even more strange, and classical Bourne script:

  ```
  countvar='1 2 3 4 5 6'
  for x in $countvar; do ... ; done
  ```

  Yields only nasty surprise.

  Here are two alternates on how to do it:

  ```
  countvar='1 2 3 4 5 6'
  for x in $(ifssplit $countvar); do ... ; done

  countvar=(1 2 3 4 5 6)
  for x in $(elements $countvar); do ... ; done
  ```
- Conditional substitution forms are supported:

  ```
  ${variable:=value}
  ${variable:-value}
  ${variable:+value}
  ```
  But appear to be borken ??
- Patterns (in case labels) are parsed once, the first time they are encountered.

  This is like with PERL's "m/../o" patterns.

- At the end of a case label, the sequentially next case labels of the same case statement will be tried for successful pattern matching (and the corresponding case label body executed). The only exceptions (apart from encountering a return statement) are:

  `again`

  : a function which retries the current case label for a match

  `break`

  : continues execution after the current case statement

- A regular expressions using variant of "`case`", with two flavours:

  `ssift`

  : A "String Shift" where the input string is handled as is.

  `tsift`

  : A "Token Shift" where the input string is spliced according to RFC-822 tokenization rules. Especially RFC-822 special characters cause tokens to split.

    With "`tsift`" the "`.`" (dot) will match any single "rfc822-token", that is, input string "foo.bar" has three tokens: "foo" (atom), "." (dot, special), and "bar" (atom).

  Overall usage of these "`sifts`" is very much like that of "`case`", including the need for matching termination tokens:

  ```
  ssift "$invar" in
  pattern
        statements
        ;;
  tfiss
  tsift "$invar" in
  pattern
        statements
        ;;
  tfist
  ```

- Various standard Bourne shell functions do not exist built in.
- The general form of function calls in the system is:

  ```
  $(funcname arguments)
  ```

  It returns a scalar or list object, and the result can be stored into variables at will.
- Relations, and other database lookups are constructed as function calls where the relation name is the function name. More about this later.

There are currently only three entry points (i.e. magic names known to the **router** code) in the configuration scripts, namely the `process`, the `router`, and the `crossbar` -functions.

- The `process()` script function is called with a file name as argument. The file is typically located in the `$POSTOFFICE/router/` directory. The `process()` is a protocol switch function which uses the form of the file name to determine how to process different types of messages.

- The `router()` script function is called with an address as argument, and returns a quad of (channel, host, user, attribs) as three separate values, corresponding to *the channel* the message should be sent out on (or, the `router` function can also be called to check on who sent a message), *the host* or node name for that channel (semantics depend per what channel is in effect), and *the address* the receiving agent should transmit to. *The fourth parameter* is "attribute" storage *variable name* from which a "privilege" value-pair is picked for recipient address security control functions.
- The `crossbar()` function is in charge of rewriting envelope addresses, selecting message header address munging type (a function to be called with each message header address), and possibly doing per-message logging or enforcing restrictions deemed necessary. It takes a sender-quad and a receiver-quad as arguments (eight parameters altogether). It returns the new values for each element of the two quads, and in addition a function name corresponding to the function to be used to rewrite header addresses for the specific destination. If the destination is to be ignored, returning a null function name will accomplish this.

There is a fourth script entrypoint used by the **smtpserver** program, namely the `server()`, which is used to implement smtpserver's realtime support facilities for "**`EXPN`**", and "**`VRFY`**" commands, and optionally also to process addresses in "**`MAIL FROM:`**`<...>`", and "**`RCPT TO:`**`<...>`" commands.

The **router** has several built in (C coded) functions. Their calling sequence and interface specification is exactly the same as for the functions defined in the configuration file. Some of these functions have special semantics, and they fall into three classes, as follows:

- Functions that are critical to the proper functioning of the configuration file interpreter:

  `return`

  returns its argument(s) as the value of a function call

  `again`

  repeats the current `case`, and `*sift` label

  `break`

  exits `case`, and `*sift` statements

- Functions that are necessary to complete the capabilities of the interpreter:

  `relation`

  defines a database to the database lookup mechanism

  `sh`

  an internal function which runs its arguments as **/bin/sh** would

- Non-critical but recommended functions:

  `echo`

  emulates **/bin/echo**

  `exit`

  aborts the **router** with the specified status code `hostname`

internal function to get and set the local idea about the system name

`trace`

turns on selected debugging output

`untrace`

turns off selected debugging output

`[`
`test`

emulates a subset of "**/bin/test**" (a.k.a. "**/bin/[**") functionality.

The *`relation`* function is described in "*Databases*", at section Section 13.2. Functions *`trace`*, and *`untrace`* are described in connection with debugging.

See *Logging and Statistics*, section Chapter 16. (*This will probably change to Reference/Router/Debugging*)

The *`hostname`* function requires some further explanation. It is intended to emulate the BSD UNIX **/bin/hostname** functionality, except that setting the hostname will only set the **router**'s idea of the hostname, not the system's. Doing so will enable generation of "Message-Id:" and "Received:" "trace" headers on all messages processed by the **router**.

It is done this way since the **router** needs to know the official domain name of the local host in order to properly generate these headers, and this method is cleaner than reserving a magic variable for the purpose.

The **router** cannot assume the hostname reported by the system is a properly qualified domain name, so the configuration file may generate it using whichever method it chooses.

If the hostname indeed is a fully qualified domain name, then:

```
hostname "hostname"
```

will enable generation of trace headers.

Finally, note that a symbol can have both a function-value and a string-value. The string value is of course accessed using the "**`$`**" prefix convention of the Bourne shell language.

To test the configuration or routing data, proceed as shown in figure Figure 13-2.

Figure 13-2. Example of running tests on router

```
sh$ $MAILBIN/router -i          (select interactive mode)
z$ rtrace                         (turn tracing on)
z$ router user@broken.address   (the address that gave you trouble)
z$ router another@address       (and so on)
```

Old salts can use "**`/usr/lib/sendmail -bt`**" instead of "**`router -i`**". Once satisfied that routing works, command:

```
zmailer router
```

will restart the **router**.

You can also run the **router** directly on a message. Copy your message to someplace other than the postoffice (`/tmp/` is usually good), to a numeric file name. If the file name is "`123`", you run

```
  $MAILBIN/router 123
```

this will create the file "`.123`" containing the control information produced by the **router**.

13.2. Databases

```
FIXME:
 - Intro
 - How 'dbases.conf' file works
 - How the databases are defined in the deep down inside ('relation' function)
 - How lookup works
```

Many of the decisions and actions taken by configuration file code depend on the specifics of the environment the MTA finds itself in. So, not just the facts that the local host is attached to (say) the UUCP network and a Local Area Network are important, but it is also essential to know the specific hosts that are reachable by this method. Hardcoding large amounts of such information into the configuration file is not practical. It is also undesirable to change what is really a program (the configuration file), when the information (the data) changes.

The desirable solution to this data abstraction problem is to provide a way for the configuration file programmer to manage such information externally to ZMailer, and access it from within the **router**. The logical way to do this is to have an interface to externally maintained databases. These databases need not be terribly complicated; after all the simplest kind of information needed is that a string is a member of some collection. This could simply correspond to finding that string as a word in a list of words.

However, there are many ways to organize databases, and the necessary interfaces cannot be known in advance. The **router** therefore implements a framework that allows flexible interfacing to databases, and easy extension to cover new types of databases.

To use a database, two things are needed: the name of the database, and a way of retrieving the data associated with a particular key from that database. In addition to this knowledge, the needs of an MTA do include some special processing pertinent to its activities and the kind of keys to be looked up.

Specifically, the result of the data lookup can take different forms: one may be interested only in the existence of a datum, not its value, or one may be looking up paths in a *pathalias* database and need to substitute the proper thing in place of "`%s`" in the string returned from the database lookup. It should be possible to specify that this kind of postprocessing should be carried out in association with a specific data access. Similarly, there may be a need for search routines that depend on the semantics of keys or the retrieved data. These possibilities have all been taken into consideration in the definition of a `relation`. A `relation` maps a key to a value obtained by applying the appropriate lookup and search routines, and perhaps a postprocessing step, applied to a specified database that has a specified access method.

The various attributes that define a `relation` are largely independent. There will of course be dependencies due to the contents or other semantics of a database. In addition to the features mentioned, each relation may optionally have associated with it a subtype, which is a string value used to tell the lookup routine which table of several in a database one is interested in.

There are no predefined relations in the **router**. They must all be specified in the configuration file before first use. This is done by calling the special function `relation` with various options, as indicated by the usage strings printed by the `relation` function when called the wrong way. See figure Figure 13-3.

Figure 13-3. "Usage:" of `relation` function

relation [-t *dbtype* [,*subtype*]] [-f *file*] [-e#] [-s#] [-bilmnpu%] [-d *driver*] [-C *configfile*] *name*

Where dbtypes include: incore, header, unordered, ordered, hostsfile, bind, selfmatch, ndbm, gdbm, btree, bhash

The "`-t`" option specifies one of several predefined database types, each with their specific lookup routine. It determines a template for the set of attributes associated with a particular relation. The predefined database types are:

`bhash`

the database is in BSD/SleepyCat DB HASH format.

`bind`

the database is the BIND nameserver, accessed through the standard resolver routines.

`btree`

the database is in BSD/SleepyCat DB BTREE format.

`dbm`

the database is in DBM format. Note that the original dbm had no `dbm_close()` function, thus there was no way to dissociate active database from a process. A bit newer variant of dbm has the close function, and multiple dbm's can be used. (You propably won't encounter this beast at all..)

`gdbm`

the database is in GNU GDBM format.

`headers`

router internal database of various headers, and how they are to be treated.

`hostsfile`

`/etc/hosts` lookup using `gethostbyname()`.

`incore`

the database is a high-speed bundle of data kept entirely in the router process core memory. This is for a short-term data storage, like handling duplicate detection.

`ldap`

Mechanism for X.500 Directory access lookup with the "Light-weight Directory Access Protocol."

`ndbm`

the database is in NDBM (new DBM) format. (At which the length of key + length of data must not exceed 1024 bytes!)

`ordered`

the database is a text file with key-datum pairs on each line, keys are looked up using a linewise binary search in the sorted file.

`selfmatch`

a special type that does translate the numerical address of format `12.34.56.78` (from within address-literal bracets) into binary form, and checks that it is (or is not) actually our own local IP addresses. This is used in address literal testing of addresses of type: `localpart@[12.34.56.78]`.

`unordered`

the database is a text file with key-datum pairs on each line, keys are looked up using a sequential search. First to match is used.

`yp`

Sun SunOS 4.x YP (these days "NIS") interface library.

A subtype is specified by appending it to the database type name separated by a slash, or a comma. For example, specifying `bind/mx` as the argument to the "`-t`" option will store "`mx`" for reference by the access routines whenever a query to that relation is processed. The subtypes must therefore be recognized by either the database-specific access routines (for translation into some other form), or by the database interface itself.

For `unordered` and `ordered` database types, the datum corresponding to a particular key may be null. This situation arises if the database is a simple list, with one key per line and nothing else. In this situation, the use of an appropriate post-processor option (e.g. "`-b`") is recommended to be able to detect whether or not the lookup succeeded.

The "`-f`" option specifies the name of the database. This is typically a path that either names the actual (and single) database file, or gives the root path for a number of files comprising the database (e.g. "`foo`" may refer to the NDBM files "`foo.pag`" and "`foo.dir`"). For the `hostsfile` type of database, the `/etc/hosts` file is the one used (and since the normal "hosts" file access routines do not allow specifying different file, this cannot be overridden).

The "`-s`" option specifies the size of the cache. If this value is non-zero (by default it is 10), then an LRU cache of this size is maintained for previous queries to this relation, including both positive and negative results.

The "`-e`" option specifies the cache data expiration time in seconds.

The "`-b`" option asks that a postprocessor is applied to the database lookup result, so the empty string is returned from the relation query if the database search failed, and the key itself it returned if the search succeeded. In the latter case, any retrieved data is discarded. The option letter is short for Boolean.

The "`-n`" option asks that a postprocessor is applied to the database lookup result, so the key string is returned from the relation query if the database search failed, and the retrieved datum string is returned if the search succeeded. The option letter is short for Non-Null.

The "`-l`" option asks that all keys are converted to *lowercase* before lookup in the database. This is mutually exclusive with the "`-u`" option.

The "`-u`" option asks that all keys are converted to *uppercase* before lookup in the database. This is mutually exclusive with the "`-l`" option.

The "`-d`" option specifies a search routine. Most commonly used argument for this option is "`pathalias`", specifying a driver that searches for the key using domain name lookup rules.

The "`-C`" option specifies a configuration file for the underlying database mechanism. Exact details depend by the database mechanisms.

The "`-%`" option enables substitution of "`%0`" thru "`%9`" patterns in the db lookup results with key, iterated partial key, or positional parameter to lookup of the database. See Reference Section 21.5.50 for more information.

Figure 13-4. Some examples of `relation` definitions

```
relation -lmt  $DBTYPE -f $MAILVAR/db/aliases$DBEXT     aliases
relation -lm%t $DBTYPE -f $MAILVAR/db/fqdnaliases$DBEXT fqdnaliases
relation -lm%t $DBTYPE -f $MAILVAR/db/routes$DBEXT -d pathalias  routes

if [ -f /etc/resolv.conf ]; then
  relation -nt bind/cname -s 100 canon # T_CNAME canonicalize hostname
  relation -nt bind/uname uname        # T_UNAME UUCP name
  relation -bt bind/mx neighbour       # T_MX/T_WKS/T_A reachability
  relation -t  bind/mp pathalias       # T_MP pathalias lookup
else
  relation -nt hostsfile -s 100 canon  # canonicalize hostname
  relation -t unordered -f $MAILBIN/db/hosts.uucp uname
  relation -bt hostsfile neighbour
  relation -t unordered -f /dev/null pathalias
fi
```

Figure 13-5. More examples of alternate forms of database reference

```
#
# We maintain an aliases database, and may access it via NDBM,
# or via indirect indexing:
#
if [ -f $MAILBIN/db/aliases.dat ]; then
    relation -t ndbm -f $MAILBIN/db/aliases aliases
else
    relation -it ordered -f $MAILBIN/db/aliases.idx aliases
fi
```

Figure 13-6. More miscellaneous `relation` definitions to illustriate various possibilities

```
relation -t unordered -f /usr/lib/news/active -b newsgroup
relation -t unordered -f /usr/lib/uucp/L.sys -b ldotsys
relation -t ordered -f $MAILBIN/db/hosts.transport -d pathalias transport
```

The final argument for the `relation` is not preceeded by an option letter. It specifies the name the relation is known under. Note that it is quite possible for different relations to use the same database (like in case of "`bind`").

Some sample relation definitions are in figure Figure 13-4. That fragment defines a set of relations that can be accessed in the same way, using the same names, independent of their actual definition.

CHECK! (`-i` option!) As the comment says, the relation name *aliases* has special significance to the **router**. Although the relation is not special in any other way (i.e. it can be used in the normal fashion), the semantics of the data retrieved are bound by assumptions in the aliasing mechanism. (Or more specifically, actually database compilation in case this isn't "ordered" or "unordered" file will handle this.)

These assumptions are that key strings are local-name's, and the corresponding datum gives a byte offset into another file (the root name of the aliases file, with a "`.dat`" extention), which contains the actual addresses associated with that alias.

The reason for this indirection is that the number of addresses associated with a particular alias can be very large, and this makes the traditional simple database formats inadequate. For example, quick lookup in a text file is only practical if it is sorted and has a regular structure. A large number of addresses associated with an alias makes structuring a problem. The situation for DBM files and variations have problems too, due to the intrinsic limits of the storage method. The chosen indirection scheme avoids such problems without loss of efficiency.

More examples on figure Figure 13-6, where the first two illustrate convenient coincidences of format, and the last definition shows what might be used if outgoing channel information is maintained in a pathalias-format database (e.g. "bar smtp!bar" means to send mail to "bar" via the SMTP channel).

13.2.1. Using a Pathalias Database With "%0" substitution

The pathalias is an UUCP era thing, and not quite what one would need these days, but just in case...

Accessing route databases is a rather essential capability for a mailer. At the University of Toronto, all hosts access a centrally stored database through a slightly modified nameserver program. If such a setup is not practical at your site, other methods are available. The most widespread kind of route database is produced by the pathalias program.

The current ZMailer can do two separate things, which were combined into the old pathalias idea:

- `relation` defines *driver* routine with "`-d pathalias`"
- `relation` defines that lookup result contained "`%0`" thru "`%9`" strings may be substituted (the "`-%`" option).

The pathalias generates key-value pairs of the form:

```
uunet    ai.toronto.edu!uunet!%s
.css.gov ai.toronto.edu!uunet!seismo!%s
```

which need to be post-processed to:

```
uunet    ai.toronto.edu!uunet!%0
.css.gov ai.toronto.edu!uunet!seismo!%0
```

which when queried about “uunet” and “beno.css.gov” correspond to the routes:

```
ai.toronto.edu!uunet
ai.toronto.edu!uunet!seismo!beno.css.gov
```

Notice that there are two basic forms of routes listed: routes to UUCP node names and routes to subdomain gateways. Depending on the type of route query, the value returned from a pathalias database lookup needs to be treated differently. For now, this may be accomplished by a configuration file relation definition and interface function as shown in figure Figure 13-7.

Figure 13-7. An example of lookup driver for genuine pathalias generated database

```
relation -t ndbm -f $MAILBIN/uuDB -d pathalias padb

# pathalias database lookup function
padblookup (name) {
    local path
    path = $(padb "$name")
    tsift "$path" in
    ((.+)!)?([^!]+)!%s
        if [ "$3" == "$name" ]; then
            path = "$2!$3"
        else
            path = "$2!$3!$name"
        fi
        ;;
    .*%s.*
        echo "illegal route in pathalias db: $path"
        ;;
    tfist
    return "$path"
}
```

This is actually a simplistic algorithm, but it does illustrate the method. The lookup algorithm used when the “`-d`” flag is specified in the `relation` definition command is rather simple; it doesn’t test various case combinations for the keys it tries. Therefore, the keys in the pathalias output data should probably be converted to a single case, and the “`-l`”, or “`-u`” option given in the `relation` definition as well..

13.2.2. Mailing Lists and `~/.forward`

FIXME! FIXME! -- VERIFY! UPDATE!

One form of mailing lists are implemented as files in the `$MAILSHARE/lists/` directory (or symlinks in there to files residing elsewere, though from a system reliability standpoint it is better to have them in that directory, and let users have symlinks to those files — consider the NFS with the user home directories in other machines. . .)

An alternate mechanism is to implement lists in the traditional sendmail manner, however it means feeding the message to the **scheduler**, and external program (**/usr/lib/sendmail**) before it comes back to the **router**.

The list *file* must have protection 0664 or stricter, as an example: 0700 has invalid bits. (ok, so the "x"-bit is not used, but illegal it is, all the same.) Preferrable protection is: 0600

The `$MAILSHARE/lists/` directory must be owned by root. The directory containing the "*aliases*" file (`$MAILSHARE/db/`) must be owned by root, and the aliases file must comply with above mentioned protections.

The owner of "*FILE*" gets "*FILE-owner*", and "*FILE-request*" mails, *unless any of the above listed limitations are breached.*

If "*FILE*" has protection 666 (for example), the ZMailer internal function "`$(filepriv $filepath)`" returns "`$nobody`" (userid of nobody), and function "`$(uid2login $nobody)`" fails, thus losing "*-owner", and "*-request" features.

Also lists with filepriv "nobody" cannot be archived simply by having an "address" of form "`/file/path`" amongst the recipient addresses.

This type of a mailing list is set up by creating a file in the `$MAILVAR/lists/` directory. The file name is the list's name (LIST) in *all lowercase* (case-insensitive matching is done by converting to lowercase before comparison).

The file contains a list of mail addresses (typically one per line) which are parsed to pull out the destination addresses. This means the users' full names can be given just as in any valid RFC822 address.

The local account which owns the file will by default receive messages sent to LIST-owner and LIST-request. This can be explicitly overridden in the aliases file. Mail to the list will go out with LIST-owner as the sender, so list bounce messages will return to the LIST-owner address. Archives of the list can be created by adding a file name address (a local pathname starting with "/") to the LIST file. The archive file is written with the ownership of the owner of the LIST file. Forwarding the mailing list into a newsgroup can be done using a mail to news script (two generations are provided in `utils/distribute` and `utils/mail2news` in the sources).

13.2.2.1. `aliases.cf` Logic

FIXME! FIXME! -- VERIFY! UPDATE!

- If an *aliases* database exists and local-part is found in it, the list of addresses mapped to by the alias entry is substituted.
- If an *mboxmap* file exists and a mapping for the local-part is found in it, the mapping (a "`host!homedir!user`" value) determines the remote recipient (`user@host`) or recipient mailbox (`homedir/../PObox/user`) if host is local.
- If local-part is a login name and a readable "`~/.forward`" file exists in the home directory, the list of addresses it contains is substituted.
- If local-part is a file basename in the `$MAILVAR/lists/` directory, the list of addresses contained in the file is substituted, and the sender address set to local-part-owner.
- If local-part is of the form "file-*owner*" or "file-*request*", where file is an entry in the `$MAILVAR/lists/` directory, the account name of the owner of the file is substituted. (File-owner identity and correct file and directory protections are important.)
- If the local-part is of format "user.name", it is optionally mapped via separate *fullnamemap*.
- If `PUNTHOST` is defined (in `/etc/zmailer.conf`) the address `local-part@$PUNTHOST` is substituted. Note that in this case the *mboxmap* mechanism should be used to ensure local spool mailbox delivery for local users.

13.2.2.2. aliases

FIXME! FIXME! -- aliases db regeneration methods have changed since

The file containing the actual aliasing data is automatically created by the **router** when asked to reconstruct the aliases database. It does this based on a text file containing the alias definitions. This text file, which corresponds to the sendmail aliases file, consists of individual alias definitions, possibly separated by blank lines or commentary. Comments are introduced by a sharp sign (octothorp: "#") at any point where a token might start (for example the beginning of a line, but not in the middle of an address), and extend to the end of the line. Each alias definition has the exact syntax of an RFC822 message header, containing an address-list, except for comments. The header field name is the local-part being aliased to the address-list that is the header value.

The fact that an alias definition follows the syntax for an RFC822 message header, introduces an incompatibility with sendmail. The string "`:include:`" at the start of a local-part (a legacy of RFC733) has special semantics. Sendmail would strip this prefix, and regard the rest of the local-part as a path to a file containing a list of addresses to be included in the alias expansion. Indeed, the **router** behaves in the same manner, but because some of the characters in the prefix are RFC822 specials, the entire local-part must be quoted. Thus, whereas sendmail(8) allowed:

```
people: :include:/usr/lib/mail/lists/people
```

the proper syntax with ZMailer is:

```
people: ":include:/usr/lib/mail/lists/people"
```

Like sendmail, if a local-part is not found in the aliases database, the **router** also checks "`~local-part/.forward`" (if such exists) for any address expansion. The `.forward` file format is also an RFC822 address list, similar to what sendmail expects.

As special cases, a local-part starting with a pipe character ("`|`") is treated as mail destined for a program (the rest of the local-part is any valid argument to a "`sh -c`" command), and a local-part starting with a slash character ("`/`") is treated as mail destined for the *file* named by the local-part.

Figure 13-8. General format of Alias file entries:

"The Key"	"The Data"

`local-address-token:`	**`"replacement address" ,`**
	`"extension line with another address"`

Protection of the aliases database must be at least 0644. Protection of the `$MAILVAR/db/` directory must be at least 03755, or stricter.

The following processing is done for (replacement) local-parts (local mail addresses): Note that `@`'s are not allowed in any local-part.

If the local-part starts with "`|`" assume it is a command specification:

```
prog-pipe: "|/path/to/program -args"
```

If the local-part starts with "`/`" assume it is a file pathname:

```
file-path: "/path/to/file"
```

If the local-part starts with "`:include:`" the rest should be a file pathname of a list of mail addresses. They are substituted:

```
included-list: ":include:/path/to/address/file"
```

After this point, all matches are case-insensitive by means of translating the value to be looked up to lower-case, and then conducting a case-sensitive lookup. *All keys in aliases et.al. must be in lower case — you can achieve this with bundled "**newaliases**" script, which calls "**makedb**" with "`-l`" option to lowercasify keys.* (The hash functions inside *ndbm/gdbm/db/dbm* are case sensitive, and as such, there is no way to avoid this requirement.)

13.2.2.3. Security Considerations

A LIST file must not be world writable, while most likely it can be group-writable. The `$MAILVAR/lists/` directory must also not be group or world writable and must be owned by root or by the owner of the LIST file. Otherwise the file is declared insecure and all addresses in the file get the least possible privilege associated (the "nobody" uid). This can cause various things to break, for example mailing list archival, or the -owner and -request features if "nobody" is not a valid account.

There is a mechanism to override using the modes on a `file/directory` as an indicator of its safeness.

Turning on the sticky bit on a file or directory tells the mailer to treat it as if it was only owner writable independent of its actual modes.

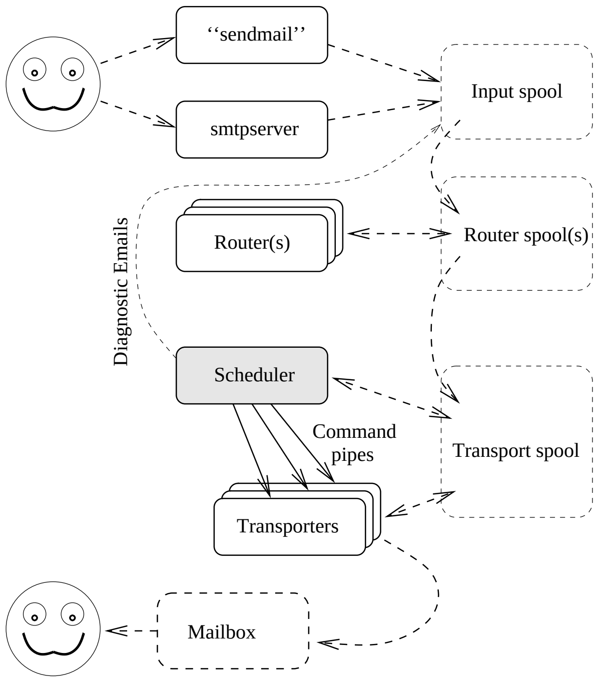
This allows `$MAILVAR/lists/` to be group or world writable and sticky-bitted if you want your general user population (or special admin group) to be able to create mailing lists.

Chapter 14. Scheduler Administration

The **scheduler** is the part of the ZMailer that manages message processing outbound from the MTA proper.

The Figure 14-1 repeats earlier picture showing central components of the system, and where the **scheduler** is in relation to them all.

Figure 14-1. ZMailer's processes; Scheduler



```
TODO!FIXME!

  Here we present longer examples out of the  scheduler.conf, and
  reasons why the default script is as it is

  - Intro
  - Principles of scheduling, "threads"
  - Clause selectors
    - "local/*"
    - "smtp/*.xyz"
    - "smtp/*"
    - "OTHER/*"
    - How to roll your own when needed
  - Something about the resource control ?
  - scheduler.auth file, and its purpose
  - MAILQv1/v2 interface, mailq-command
    - but the protocols are REFERENCE material
  - (manual-)expirer
  - Diagnostics reporting, forms files

  Appendix B contains full samples of scheduler.conf, and scheduler.auth
```

The major action of the **scheduler** is to periodically start up *Transport Agents* and tell them what to do. This is controlled by a table in a configuration file that is read by the **scheduler** when it starts.

The **scheduler** receives the messages from the **router** in *two* files: Original message file in `$POSTOFFICE/queue/`, and **router** written transport-specifications in `$POSTOFFICE/transport/`.

Because usually (UNIX) systems don't like of having very large amounts of files in directories, as lookups for them become intolerably slow, the **scheduler** has a subdirectory hashing mechanism. Both `queue/` and `transport/` *can* be split to subdirectories at one or two levels with names of "`A`" thru "`Z`". See for the "`-H`" option of the **scheduler** below.

The idea with the sub-directory hashes is to split the workset into as even subsets as possible, usually busy systems are run with "`-HH`" to have 26*26 sub-sub-directories into which the sets gets divided.

14.1. Principiles of scheduling: Threads

The **router** produces recipients "address quads" which consists of four components:

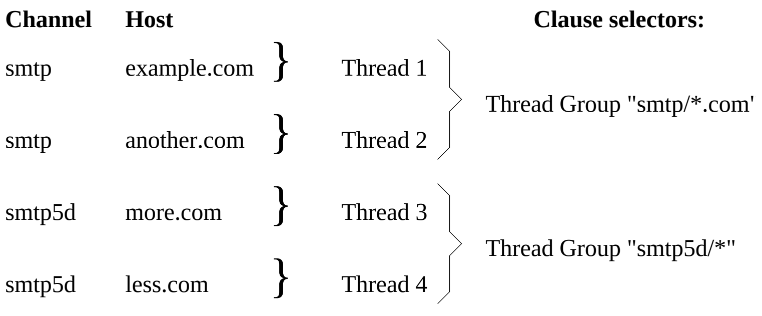
- Channel
- Host
- User
- Privilege

Of these, *Channel*, and *Host* parts are used by the **scheduler** to classify message recipients and to choose to what *Transport Agents* to use, and how to use them.

To control how messages are sent out, the **scheduler** uses catenation of: *Channel/Host* to group recipients. All recipients with *same* values there are groupped together in what is called *thread*.

Another related thing is so called *thread group*, which is the collection of all threads within same *scheduler Selector Clause*. See Section 14.3.

Figure 14-2. Scheduler's Threads/Thread Groups



In normal case the scheduling is done by running *single* transport agent for the *thread*, which delivers messages one at the time. This means that a single destination with thousands of messages does not block the system significantly more than some other destination with a single message.

Also in normal cases, when messages in given thread are all delivered, or otherwise determined that nothing can be done, the transport agent can be switched to another thread within the same *thread group*.

14.2. Scheduler Resource Control

FIXME! TO BE WRITTEN!

Note, there are three kinds of resource-pool limitation parameters which control when a given channel+host pair (thread) is NOT taken into processing:

`MaxTA`: (Set in "*/*" clause)

> GLOBAL parameter limiting the number of transport-agent processes that the scheduler can have running at the same time.
>
> With this you can limit the number of TA processes running at the same time lower than maximum allowed by your OS setup.
>
> The scheduler detects the max number of FDs allowed for a process, and analyzing how many FDs each TA interface will need -- plus reserving 10 FDs for the itself, result is "probed maxkids".

`MaxChannel`: (default: "probed maxkids")

> Selector clause specific value limiting how many transport-agent processes can be running on which the "channel" part is the same. You may specify dis-similar values for these as well. For example you may use value '50' for all your 'smtp' channel entries, except that you want always to guarantee at least five more for your own domain deliveries, and thus have:

```
smtp/*your.domain
      maxchannel=55
```

> If the sum of all "maxchannel" values in different channels exceeds that of "maxta", then "maxta" value will limit the amount of work done in extreme load situations.

`MaxRing`: (default: "probed maxkids")

> This limits the number of parallel transport agents within each selector definition. This defined the size of the POOL of transport agent processes available for processing the threads matching the selector clause.

`MaxTHR`: (default: 1)

> This limits the number of parallel transport agents within each thread; that is, using higher value than default "1" will allow running more than one TA for the jobs at the thread.
>
> Do note that running more than one TA in parallel may also require lowering OVERFEED value. (E.g. having a queue of 30 messages will not benefit from more TAs, unless they all get something to process. Having OVERFEED per default at 150 will essentially feed whole queue to one TA, others are not getting any.)

`OverFeed`:

> This tells how many job specifiers to feed to the TA when the TA process state is "STUFFING" Because the scheduler is a bit sluggish to spin around to spot active TAs, it does make sense to feed more than one task to a TA, and then wait for the results.

14.3. The `scheduler.conf` file

Any line starting with a “#” character is assumed to be a comment line, and is ignored, as are empty lines. All other lines must follow a rigid format.

The **scheduler** configuration file consists of a set of clauses. Each clause is selected by the pattern it starts with. The patterns for the clauses are matched, in sequence, with the *channel/host* string for each recipient address. When a clause pattern matches an address, the parameters set in the clause will be applied to the scheduler’s processing of that address. If the clause specifies a command, the clause pattern matching sequence is terminated. Example of the clause can be seen in figure Figure 14-3.

Figure 14-3. Example of `scheduler.conf` clause

```
local/*
        interval=10s
        expiry=3h
        # want 20 channel slots in case of blockage on one
        maxchannel=20
        # want 20 thread-ring slots
        maxring=20
        command="mailbox -8"
```

A clause consists of:

- A selection pattern (in shell style) that is matched against the channel/host string for an address.
- 0 or more variable assignments or keywords (described below).

If the selection pattern does not contain a ’/’, it is assumed to be a channel pattern and the host pattern is assumed to be the wildcard ’*’.

The components of a clause are separated by whitespace. The pattern introducing a clause must start in the first column of a line, and the variable assignments or keywords inside a clause must not start in the first column of a line. This means a clause may be written both compactly all on one line, or spread out with an assignment or keyword per line.

If the clause is empty (i.e., consists only of a pattern), then the contents of the next non-empty clause will be used.

The typical configuration file will contain the following clauses:

- a clause matching all addresses (using the pattern */*) that sets up default values.
- a clause matching the local delivery channel (usually local).
- a clause matching the deferred delivery channel (usually hold).
- a clause matching the error reporting channel (usually error).
- clauses specific to the other channels known by the **router**, for example: smtp and uucp.

The actual names of these channels are completely controlled by the **router** configuration file.

Empty lines, and lines whose first non-whitespace character is “#”, are ignored.

Variable values may be unquoted words or values or double quoted strings. Intervals (delta time) are specified using a concatenation of numbers suffixed with ’s’, ’m’, ’h’, or ’d’; modifiers designating the number as a second, minute, hour, or day value. For example: 1h5m20s.

The known variables and keywords, and their typical values and semantics are:

`interval` (1m)

specifies the primary retry interval, which determines how frequently a transport agent should be scheduled for an address. The value is a delta time specification. This value, and the retries value mentioned below, are combined to determine the interval between each retry attempt.

`idlemax`

When a transport agent runs out of jobs, they are moved to “idle pool”, and if a transport agent spends more than idlemax time in there, it is terminated.

`expiry` (3d)

specifies the maximum age of an address in the scheduler queue before a repeatedly deferred address is bounced with an expiration error. The actual report is produced when all addresses have been processed.

`retries` (1 1 2 3 5 8 13 21 34)

specifies the retry interval policy of the scheduler for an address. The value must be a sequence of positive integers, these being multiples of the primary interval before a retry is scheduled. The scheduler starts by going through the sequence as an address is repeatedly deferred. When the end of the sequence is reached, the scheduler will jump into the sequence at a random spot and continue towards the end. This allows various retry strategies to be specified easily:

brute force (or “jackhammer”)

```
retries=0
```

constant primary interval

```
retries=1
```

instant backoff

```
retries="1 50 50 50 50 50 50 50 50 50 50 50 50"
```

slow increasing (fibonacci) sequence

```
retries="1 1 2 3 5 8 13 21 34"
```

s-curve sequence

```
etries="1 1 2 3 5 10 20 25 28 29 30"
```

exponential sequence

```
retries="1 2 4 8 16 32 64 128 256"
```

`maxta` (0)

FIXME: REVISE RESOURCE CHECK NOTES! if retrying an address would cause the number of simultaneously active transport agents to exceed the specified value, the retry is postponed. The check is repeated frequently so the address may be retried as soon as possible after the scheduled retry interval. If the value is 0, a value of 1000 is used instead. Keep in mind that all running transport agents will keep open two `pipe`(2) file-handles, (or one `socketpair`(2), if system has such bidirectional pipe entity,) and thus system-wide limits may force a lower maximum than 1000. On a system with a maximum of 256 open files, this would most likely succeed at 120.

`maxchannel` (0)

if retrying an address would cause the number of simultaneously active transport agents processing mail for the same channel to exceed the specified value, the retry is postponed. The check is repeated frequently so the address may be retried as soon as possible after the scheduled retry interval. If the value is 0, a value of 1000 is used instead.

`maxring` (0)

Recipients are groupped into "threads", and similar threads are groupped into "thread-rings", where the same transport agent can be switched over from one recipient to another. This defines how many transport agents can be running at any time at the ring.

`skew` (5)

is the maximum number of retries before the retry time is aligned to a standard boundary (seconds since epoch, modulo primary interval). The lower this number (1 is lowest), the faster the alignment is done. The purpose of this alignment is to ensure that eventually a single transport agent invokation will be able to process destination addresses that arrived randomly at the scheduler.

`user` (root)

is the user id of a transport agent processing the address. The value is either numeric (a uid) or an account name.

`group` (daemon)

is the group id of a transport agent processing the address. The value is either numeric (a gid) or a group name.

`command` (smtp -srl ${LOGDIR}/smtp ${host}

is the command line used to start a transport agent to process the address. The program pathname is specified relative to the `$MAILBIN/ta/` directory. The string "${channel}" is replaced by the current matched channel, and "${host}" is replaced by the current matched host, from the destination address, and "${`LOGDIR`}" substitutes ZENV variable `LOGDIR` value there.

It is strongly recommended that the "`${host}`" is not to be used on command definition, as it limits the recyclability of the idled transporter.

`bychannel`

is a keyword (with no associated value which tells the **scheduler** that the transport agent specified in the command will only process destination addresses that match the first destination channel it encounters.

This is automatically set when the string "`${channel}`" occurs in the command, but may also be specified manually. This is rarely used.

`queueonly`

a clause with queueonly flag does not auto-start at the arrival of a message, instead it must be started by means of **smtpserver**(8) command ETRN thru an SMTP connection.

An example of full `scheduler.conf` file is in figure Figure 14-4.

Figure 14-4. Example of full `scheduler.conf` file

```
# Default values
*/*       interval=1m expiry=3d retries="1 1 2 3 5 8 13 21 34"
          maxring=0 maxta=0 skew=5 user=root group=daemon
# Boilerplate parameters for local delivery and service channels
local/*   interval=10s expiry=3h maxchannel=2 command=mailbox
error     interval=5m maxchannel=10 command=errormail
hold/*    interval=5m maxchannel=1 command=hold

# Miscellaneous channels supported by router configuration
smtp/*.toronto.edu
smtp/*.utoronto.ca maxchannel=10 maxring=2
          command="smtp -srl ${LOGDIR}/smtp"
smtp      maxchannel=10 maxring=5
          command="smtp -esrl ${LOGDIR}/smtp"

uucp/*    maxchannel=5 command="sm -c ${channel} uucp"
```

The first clause (*/*) sets up default values for all addresses. There is no command specification, so clause matching will continue after address have picked up the parameters set here.

The third clause (error) has an implicit host wildcard of "*", so it would match the same as specifying *error/** would have.

The fifth clause (*smtp/*.toronto.edu*) has no further components so it selects the components of the following nonempty clause (the sixth).

Both the fifth and sixth clauses are specific to address destinations within the TORONTO.EDU and UTORONTO.CA organization (the two are parallel domains). At most 10 deliveries to the *smtp* channel may be concurrently active, and at most 2 for all possible hosts within TORONTO.EDU. If "$host" is mentioned in the command specification, the *transport agent* will only be told about the message control files that indicate SMTP delivery to a particular host. The actual host is picked at random from the current choices, to avoid systematic errors leading to a deadlock of any queue.

14.4. Scheduler's Mailq

FIXME! TO BE WRITTEN!

14.5. Scheduler's `scheduler.auth` control file

FIXME! TO BE WRITTEN!

14.6. manual-expirer

FIXME! TO BE WRITTEN!

14.7. Scheduler’s Diagnostics Reporting

FIXME! TO BE WRITTEN!

14.7.1. Scheduler’s Diagnostics Reporting, Forms Files

FIXME! TO BE WRITTEN!

Chapter 15. Transport Agent Administration

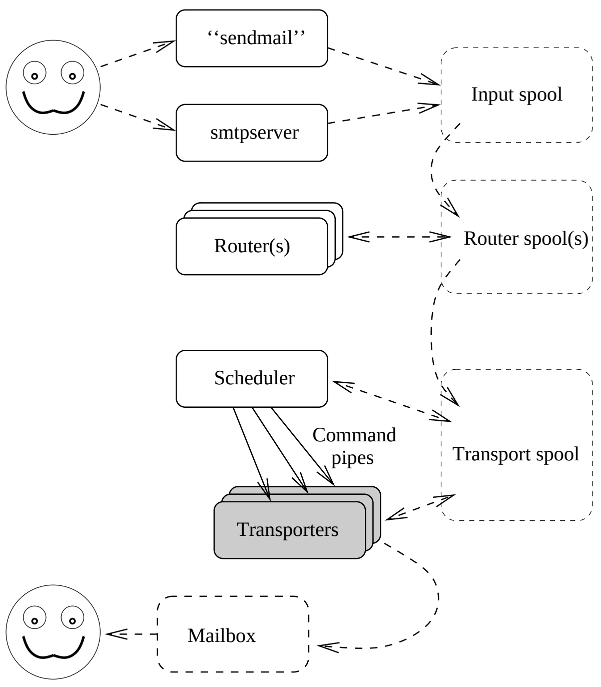
```
- Move to be a SECTION of Scheduler ?
  - These are support thingies for the Scheduler anyway.
- Less details than REFERENCE has, focus differently

- sm
- smtp
- mailbox
- hold
- expirer
- errormail
```

These are ZMailer's components driven by the **Scheduler** to actually do message delivery actions.

The Figure 15-1 repeats earlier picture showing central components of the system, and where the *transport agents* are in relation to the whole.

Figure 15-1. ZMailer's processes; Transport Agents



15.1. Sm Transport Agent

sm is a ZMailer's **sendmail**(8) compatible *transport agent* to deliver messages by invoking a program with facilities and in a way compatible with a **sendmail**(8) MTA.

The program scans the message control files named on stdin for addresses destined for the channel and/or the host given on the command line. If any are found, all matching addresses and messages are processed according to the specifications for the mailer in the configuration file.

The exit status of a mailer should be one of the standard values specified in *#include <`sysexits.h`>*. Of these, EX_OK indicates successful deliver, and EX_DATAERR, EX_NOUSER, EX_NOHOST, EX_UNAVAILABLE, and EX_NOPERM indicate permanent failure. All other exit codes will be treated as a temporary failure and the delivery will be retried.

Usage:

sm [`-8`] [`-H`] [`-Q`] [`-V`] [`-f` *`configfile`*] -c *`channel`* `-h` *`host mailer`*

Configuration:

The configuration file `$MAILSHARE/sm.conf` associates the mailer keyword from the command line with a specification of a delivery program. This is very similar to the way the definition of a “mailer” in **sendmail**(8). It requires flags, a program name, and a command line specification. These are in fact the fields of the entries of the configuration file. Lines starting with whitespace or a “#” are ignored, and all others are assumed to follow format shown in figure Figure 15-2.

Figure 15-2. Sample `sm.conf` file

```
#mailer flags program        argument list
#====== ===== ============== ================================
local   mS    sm/localm      localm -r $g $u
prog    -     /bin/sh        sh -c $u
tty     rs    /usr/local/to  to $u
uucp    U     /usr/bin/uux   uux - -r -a$g -gC $h!rmail ($u)
usenet  m     sm/usenet      usenet $u
test    n     sm/test        test $u
```

The mailer field extends from the beginning of the line to the first whitespace. It is used simply as a key index to the configuration file contents. One or more whitespace is used as the field separator for all the fields.

The flags field contains a concatenation of one-letter flags. If no flags are desired, a “`-`” character should be used to indicate presence of the field. All normal sendmail (of 8.11(.0)) flags are recognized, but the ones that do not make sense in the context of ZMailer will produce an error (or some are ignored). The flags that change the behaviour of **sm** are:

`b`

will activate BSMTP-type wrapping with a “hidden-dot” algorithm; e.g. quite ordinary SMTP stream, but in “batch mode”.

`B`

The first “`B`” turns on similar BSMTP wrapping as “`b`”, but adds SIZE and, if the **sm** is started with option “`-8`”, also 8BITMIME options. The second “`B`” adds there also DSN (Delivery Status Notification) parameters.

`E`

will prepend “>” to any message body line starting with “*From* ”. (Read: “From-space”)

`f`

adds “`-f sender`” arguments to the delivery program.

`n`

will not prepend a “*From* ”-line (normal mailbox separator line) to the message.

`r`

adds “`-r sender`” arguments to the delivery program.

`S`

will run the delivery program with the same real and effective uid as the **sm** process. If this flag is not set, the delivery program will be run with the real uid of the **sm** process. This may be useful if **sm** is setuid.

`m`

informs **sm** that each instance of the delivery program can deliver to many destinations. This affects `$u` expansion in the argument list, see below.

`P`

prepends a “Return-Path:” header to the message.

`U`

will prepend a “*From* ”-line, with a “remote from myuucpname” at the end, to the message. This is what is expected by remote **rmail**(1) programs for incoming UUCP mail.

`R`

use CRLF sequence as end-of-line sequence. Without it, will use LF-only end-of-line sequence.

`X`

does SMTP-like “hidden-dot” algorithm of doubling all dots that are at the start of the line.

`7`

will strip (set to 0) the 8th bit of every character in the message.

The path field specifies the location of the delivery program. Relative pathnames are allowed and are relative to the `$MAILBIN/` directory.

The arguments field extends to the end of the line. It contains whitespace separated `argv` parameters which may contain one of the following sequences:

`$g`

which is replaced by the sender address.

`$h`

which is replaced by the destination host.

`$u`

which is replaced by the recipient address. If the “`m`” mailer flag is set and there are several recipients for this message, the argument containing the “`$u`” will be replicated as necessary for each recipient.

Chapter 16. Logging and Statistics for Administrator

```
What material is applicable here ?

How to control logging ?
 - Settings/variables/whatnot controlling it

How to rotate logfiles ?
 - Example scripts for rotation ?
```

IV. Reference

Here are *reference* versions of the documentation for each subsystem, these go very deep into details.

Chapter 17. Smtpserver Reference

The ZMailer distribution contains an **smtpserver** program for the BSD socket implementation of TCP/IP. It is an asynchronous implementation, in that while address syntax is checked in real time, semantics are not, nor are other (optional in the SMTP standard) functions that require **router** functionality.

The server will run an RFC-821 syntax scanner for addresses, plus possible policy analysis phase, and if things are ok, it says "*Yes yes, sure!*" to everything. The program may also be used in non-daemon mode to unpack BSMTP format messages on the standard input stream.

This program implements the server side of the *SMTP* protocol as described in RFC821, and knows about the common extensions to the protocol expected by sendmail and BSMTP clients. By default the program will kill the previous **smtpserver** daemon, if any, then detach and listen for *SMTP* connections. Incoming messages will be submitted for processing using the *zmailer(3)* interface to ZMailer. Some non-trivial address checking is doable in optional policy analysis functions within the **smtpserver**, or can be acomplished with synchronous (or asynchronous) running of router. This behaviour can be changed by a command line option (or HELO/EHLO style patterns) if you cannot afford to transfer data just to bounce it back.

All **router** assisted checking is done by executing the **router**(8) program in interactive mode, and executing well-known shell function with well-known parameters for each request.

The server implements also most of the ESMTP facilities invented up to date (Feb, 2000). The ones that are active are visible at greeting response to "*EHLO*" command, as can be seen in figure Figure 17-1.

Figure 17-1. Sample "*EHLO*" greeting with smtpserver

```
$ telnet 127.1 smtp
Connected to 127.1.
Escape character is '^]'.
220 localhost ZMailer ....
EHLO foo
250-localhost expected "EHLO localhost"
250-SIZE 1234567
250-8BITMIME
250-PIPELINING
250-CHUNKING
250-ENHANCEDSTATUSCODES
250-EXPN
250-VRFY
250-DSN
250-X-RCPTLIMIT 10000
250-ETRN
250 HELP
...
```

17.1. Smtpserver Runtime Parameters

Usage:

smtpserver [-46aginvBV] [-p `port`] [-l `logfile`] [-s `'strict'`] [-s `[ftveRS]`] [-L `maxloadaver`] [-M `SMTPmaxsize`] [-P `postoffice`] [-R `router`] [-C `cfgfile`]

Parameters:

`-4`

Explicitly use IPv4 type of socket even on machines that are capable to do IPv6 type of sockets.

`-6`

Explicitely (try to) use IPv6 type of socket even if the machine does not support it. By default the server will try to use IPv6, if it has been compiled in an environment where it is present, but will fall back to IPv4, if the runtime system does not have IPv6.

`-8`

This option is part of optional inbound translate processing; see "`-X`" option below.

`-a`

Turn on RFC931/RFC1413 identification protocol, and log the information acquired with it into the submitted file. Reliability, validity, worthwhileness, ... all such are suspect at this.

`-B`

Flags the email to arrive via BSMTP channel (via BITNET, for example).

`-B cfgfile`

Specifies nonstandard configuration file location; the default is `$MAILSHARE/smtpserver.conf`.

`-d nnnn`

This option sets numeric debug value. Any non-zero will work. This numeric argument is provision for possible bit-flag or level oriented debugging mode....)

`-g`

The "*gullible*" option will make the program believe any information it is told (such as origin of a connection) without checking.

`-h`

Check "**`HELO`**" parameter very closely (syntax), and if it is bad, complain with "501". Such behaviour is against interoperability minded "*Be lenient on what you accept*" policy, and apparently will break a lot of *common* clients....

`-i`

Runs the server interactively, which makes it usable for processing a batched *SMTP* stream (*BSMTP*) on `stdin`. With “`-v`” option this echoes incoming BSMTP to create more accurate faximile of BITNET BSMTP mailers.

`-L` *`maxloadaver`*

The maximum load-average the system is under when we still accept email.

Not all systems are supported for load-aver extraction; and as that information happens to be very poorly extractable without major magics, it is our considered opinnion that it is better to spend time to figure other methods for limiting incoming email induced load impact, than trying to see any load-average — anyway the ZMailer smtpserver*used without interactive routing is very low load inpact system.*

`-l 'SYSLOG'`

Specifies that incoming *SMTP* conversations are logged via `syslog(3)` to system **syslogd(8)** server by using facility `LOG_MAIL` and level `LOG_DEBUG` messages.

`-l` *`LOGFILE`*

Specifies a logfile and enables recording of incoming *SMTP* conversations to go there.

This can be used in parallel with `-l 'SYSLOG'` version!

`-M` *`SMTPmaxsize`*

Defines the absolute maximum size we accept from incoming email. (Default: infinite) (This is a local policy issue.)

`-n`

Indicates the program is being run from {\em inetd(8)}.

`-P` *`postofficedir`*

Specifies an alternate `$POSTOFFICE/` directory.

`-P` *`port`*

Specifies the TCP port to listen on instead of the default *SMTP* port: 25.

`-R` *`routerpath`*

Specifies an alternate **router**(8) program to use for address verification.

`-s 'strict'`

This turns server protocol processing into extremely strict mode, any misplaced character is rejected. Not very practical in real file, but nice for protocol interoperability bakeout testing.

`-s 'strict'`

Specifies the style of address verification to be performed. There are four independent commands that can invoke some kind of address verification, and four independent flags to control whether this should be done.

They are:

`f`

Run “**`MAIL FROM`**” address through online router for analysis.

`t`

Run “**`RCPT TO`**” address through online router for analysis.

`v`

Enable “**`VRFY`**” command for this style selector (if configuration “*`PARAM vrfycmd`*” is in effect)

`e`

Enable “**`EXPN`**” command for this style selector (if configuration “*`PARAM expncmd`*” is in effect)

`R`

Require incoming addresses to be of fully-qualified domained form.

Don’t use this if you want to allow non-domained addresses accepted into your server through SMTP.

`S`

Allow “*Sloppy*” behaviour from the sending smtp clients; namely allow “**`MAIL FROM:foo@bar`**”, that is, an addresses *without* mandatory (RFC-821) angle braces.

The flags are concatenated to form the argument to the “`-s`” option. The default is “`ve`”.

`-S` *`suffixstyle`*

This defines log suffix which can alter the default logfile name to one which splits incoming traffic into separate files.

Possible values are:

`'remote'`

Append remote hostname to the logfile name (after a dot) so that from host “foo.bar.edu” the logfile would be: “smtpserver.foo.bar.edu”.

`'local'`

Append local end reversed hostname to the logfile name (after a dot) so that in multihomed hosts all different “hosts” have different logfiles. Such does, of course, assume that different IP addresses in the host reverse to different names.

`-v`

This is a “*verbose*” option to be used with “`-i`” option. This is especially for “BSMTP” processing.

`-V`

prints a version message and exits.

`-X`

This is "*Xlate*" option. For more info, see source file:

"It may be necessary in some cases (e.g. in Cyrillic-language countries) to translate charset on the messages coming from the clients with, e.g. old Eudora or other MUAs that do not correctly support koi8-r charset. ..."

—`README.translation`

17.2. Smtpserver Configuration

If the `$MAILSHARE/smtpserver.conf` file exists it is read to configure two kinds of things. Specifically the following:

PARAM -entries

Allow server start-time parametrization of several things, including:

- system parameters
- help texts
- acceptance/rejection database definitions

The style (`-s`) options

Behaviour is based on glob patterns matching the **`HELO`/`EHLO`** name given by a remote client. Lines beginning with a "#" or whitespace are ignored in the file, and all other lines must consist of two tokens: a shell-style (glob) pattern starting at the beginning of the line, whitespace, and a sequence of style flags. The first matching line is used.

As a special case, the flags section may start with a "!" character in which case the remainder of the line is a failure comment message to print at the client. This configuration capability is intended as a way to control misbehaving client software or mailers.

17.2.1. Smtpserver configuration; PARAM -entries

PARAM maxsize *`nn`*

Maximum size in the number of bytes of the entire spool message containing both the transport envelope, and the actual message. That is, if the max-size is too low, and there are a lot of addresses, the message may entirely become undeliverable..

This sets system default value, and overrides commandline "`-M`" option.

PARAM max-error-recipients *`nn`*

In case the message envelope is an error envelope (MAIL FROM:<>), the don't accept more than this many separate recipient addresses for it. The default value is 3, which should be enough for most cases. (Some SPAMs claim to be error messages, and then provide a huge number of recipient addresses... Of course as spam-spewers learn, they begin just sending single recipients per message — less efficient, but working still...)

PARAM MaxSameIpSource *`nn`*

(Effective only on daemon-mode server — not on "`-i`", nor "`-n`" modes.) Sometimes some systems set up multiple parallel connections to same host (qmail ones especially, not that ZMailer has entirely clean papers on this either — at least up to 2.99.X series), we won't accept more than this many connections from the same IP source address open in parallel.

The default value for this limit is 10.

The principal reason for this has been authors experience at nic.funet.fi, where some MS-Windows users have caused *huge* numbers of parallel connections to some services. So huge, that the system did in fact run out of swap due to that, and caused all manner of other nasty things to occur too... All this simply because some windows client had opened 800+ parallel sessions, and each server process consumed separate blocks of swap space... To avoid the system to succumb under such an accidental denial-of-service attack at the overall system, this parallel limit was created.

PARAM TcpRcvBufferSize *`nn`*

This sets `setsockopt(SO_RCVBUF)` value, in case the system default is not suitable for some reason.

PARAM TcpXmitBufferSize *`nn`*

This sets `setsockopt(SO_SNDBUF)` value, in case the system default is not suitable for some reason.

PARAM ListenQueueSize *`nn`*

This relates to newer systems where the `listen(2)` system call can define higher limits, than the traditional/original 5. This limit tells how many nascent TCP streams we can have in SYN_RCVD state before the socket stops answering to incoming SYN packets requesting opening of a connection. Such sockets have *not* opened sufficiently to reach a state where bi-directional communication has been established, thus they won't appear to `accept(2)` yet for the server to pick them up!

There are entirely deliberate denial-of-service attacks based on flooding to some server many SYNS on which it can't send replies back (because the target machines don't have network connectivity, for example), and thus filling the back-queue of nascent sockets.

This can also happen accidentally, as the connectivity in between the client host, and the server host may have a black hole into which the SYN-ACK packets disappear, and the client thus will not be able to get the TCP startup three-way handshake completed.

Most modern systems can have this value upped to thousands to improve systems resiliency against malicious attacks, and most likely to provide complete immunity against the accidental "attack" by the failing network routing.

Do consult your system specific information on how much memory each nascent (and matured) socket will require before you commence upping this very much. You might commit heaps of unswappable memory to useless waste.

PARAM help *`string`*

This one adds yet another string (no quotes are used) into those that are presented to the client when it asks for "**`HELP`**" in the SMTP session.

PARAM debugcmd

Enables "`DEBUG`" command in the server. This command turns on various trace functions which ruin the protocol from standards compliant client, but may help interactive debuggers.

PARAM expncmd

Enables "`EXPN`" command in the server.

PARAM vrfycmd

Enables "`VRFY`" command in the server.

PARAM PolicyDB *`dbtype dbpath`*

This defines the database type, and file path prefix to the binary database containing policy processing information. Actual binary database file names are formed by appending type specific suffixes to the path prefix. For example NDBM database appends ".pag" and ".dir", while BSD-Btree appends only ".db". (And the latter has only one file, while the first has two.)

More information below, and at **newdb** at Section 24.3

PARAM allowsourceroute

When present, this parameter will not convert input of form <@aa,@bb:cc@dd> into source-route-less form of <cc@dd> Instead it carries the original source-route into the system as is.

A possible **smtpserver** configuration file is shown in figure Figure 17-2.

Figure 17-2. Full-featured `smtpserver.conf` file example

```
#
# {\rm{}smtpserver.conf - autogenerated edition}
#
#PARAM maxsize              10000000    # {\rm{}Same as -M -option}
#PARAM max-error-recipients        3    # {\rm{}More than this is propably SPAM!}
#PARAM MaxSameIpSource            10    # {\rm{}Max simultaneous connections from}
#                                       # {\rm{}any IP source address}
#PARAM TcpRcvBufferSize        32000    # {\rm{}Should not need to set!}
#PARAM TcpXmitBufferSize       32000    # {\rm{}Should not need to set!}
#PARAM ListenQueueSize            10    # {\rm{}listen(2) parameter}

# {\rm{}Enables of some commands:}
PARAM debugcmd
PARAM expncmd
PARAM vrfycmd

PARAM help =============================================================
PARAM help  This mail-server is at Yoyodyne Propulsion Inc.
PARAM help  Our telephone number is: +1-234-567-8900, and
PARAM help  telefax number is: +1-234-567-8999
PARAM help  Our business-hours are Mon-Fri: 0800-1700 (Timezone: -0700)
PARAM help
PARAM help  Questions regarding our email service should be sent via
PARAM help  email to address  <postmaster@OURDOMAIN>
PARAM help  Reports about abuse are to be sent to: <abuse@OURDOMAIN>
PARAM help =============================================================

# {\rm{}Uncomment following for not to strip incoming addresses of format:}
# <@aa,@bb:cc@dd>  into non-source-routed base form: <cc@dd>
#PARAM  allowsourceroute

PARAM   accept-percent-kludge # "localpart" can contain '%' and '!'
#PARAM  reject-percent-kludge # "localpart" can't contain  --"--

# {\rm{}The policy database:  (NOTE: See  'makedb'  for its default suffixes!)}
PARAM  policydb   btree  /opt/mail/db/smtp-policy

#
# HELO/EHLO-pattern     style-flags
#               [max loadavg]

localhost           999 ftveR
some.host.domain    999 !NO EMAIL ACCEPTED FROM YOUR MACHINE

# {\rm{}If the host presents itself as:  HELO [1.2.3.4], be lenient to it..}
# {\rm{}The syntax below is due to these patterns being SH-GLOB patterns,}
# {\rm{}where brackets are special characters.}

\[*\]               999 ve

# {\rm{}Per default demant strict syntactic adherence, including fully}
# {\rm{}qualified addresses for  MAIL FROM, and RCPT TO.  To be lenient}
# {\rm{}on that detail, remove the "R" from "veR" string below:}

*                   999 veR
```

17.3. policy-builder.sh utility

The policy database that {\em smtpserver} uses is built with {\tt policy-builder.sh} script, which bundles together a set of policy source files:

```
  DB/smtp-policy.src   The boilerplate
  DB/localnames        ('= _localnames')
  DB/smtp-policy.relay ('= _full_rights')
  DB/smtp-policy.mx    ('relaytargets +')
  DB/smtp-policy.spam  ('= _bulk_mail')
```

```
At the moment, {\tt smtp-policy.spam} source is retrieved with LYNX from
the URL:
\begin{alltt}\medskip\scriptsize\medskip
http://www.webeasy.com:8080/spam/spam_download_table
\medskip\end{alltt}\medskip
however it seems there are sites out there that are spam havens, and
that serve valid spam source/responce domains, which are not registered
at that database.

{\em If you want, you can modify your {\tt smtp-policy.src} boilerplate
file as well as your installed {\tt\small policy-builder.sh} script.}
{\bf In fact you SHOULD modify both to match your environment!}
```

```
Doing {\tt make install} will overwrite {\tt\small policy-builder.sh},
but not {\tt smtp-policy.src}.

Basically these various source files (if they exist) are used to
combine knowledge of valid users around us:

\begin{description}
\item[\tt localnames] \mbox{}

Who we are -- ok domains for receiving.

\item[\tt smtp-policy.relay] \mbox{}

Who can use us as outbound relay.

Use  {\em\verb/[/ip.number\verb/]//maskwidth}  here for
listing those senders (networks) we absolutely trust.
You may also use domains, or domain suffixes so that the IP-reversed
hostnames are accepted (but that is a it risky thing due to ease of
fakeing the reversed domain names):

\begin{alltt}\medskip\hrule\medskip
[11.22.33.00]/24
ip-reversed.host.name
.domain.suffix
\medskip\hrule\end{alltt}\medskip

Server sets its internal "always_accept" flag at the source IP tests
before it decides on what to tell to the contacting client.
The flag is not modified afterwards during the session.

Usage of domain names here is discouraged as there is no way to tell
that domain "foo.bar" here has different meaning than same domain
elsewere -- at "{\tt smtp-policy.mx}", for example.

\item[\tt smtp-policy.mx] \mbox{}

Who really are our MX clients.
Use this when you really know them, and don't want just to trust
that if recipient has MX to you, it would be ok...

You can substitute this knowledge with a fuzzy feeling by using
"acceptifmx -" attribute at the generic boilerplate.
List here domain names.
\begin{alltt}\medskip\hrule\medskip
 mx-target.dom
 .mx-target.dom
\medskip\hrule\end{alltt}\medskip

You CAN also list here all POSTMASTER addresses you accept email routed to:

\begin{alltt}\medskip\hrule\medskip
 postmaster@local.domain
 postmaster@client.domain
\medskip\hrule\end{alltt}\medskip

these are magic addresses that email is accepted to, even when everything
```

```
else is blocked.

\item[\tt smtp-policy.spam] \mbox{}

Those users, and domains that are absolutely no-no for senders,
or recipients no matter what earlier analysis has shown.
(Except for those senders that we absolutely trust..)

\begin{alltt}\medskip\hrule\medskip
 user@domain
 user@
 domain
\medskip\hrule\end{alltt}\medskip

The “{\tt policy-builder.sh}” builds this file from external sources.

\end{description}
```

17.4. Relaying Control Policy Language

```
Policy based filter database boilerplate for smtp-server.

File:  {\tt \$MAILVAR/db/smtp-policy.src}

This file is compiled into an actual database using the command:
\begin{alltt}\medskip\hrule\medskip
  \$MAILBIN/policy-builder.sh
\medskip\hrule
\end{alltt}\par


The basic syntax of non-comment lines in the policy source is:
\begin{alltt}\medskip\hrule\medskip
  key  [attribute value]* [= _tag]
\medskip\hrule
\end{alltt}\par

There are any number of attribute-value pairs associated with the key.

There can be only one key of any kind currently active, unless “{\em makedb}”
is called with “-A” option (Append mode) in which case latter appearances
of some keys will yield catenation of of latter data into previous datasets.
(This may or may not be a good idea...)

The key can be any of following forms:
\begin{description}
\item[\rm domain, or .domain.suffix] \mbox{} \\
a domain name optionally preceded by a dot (.)

\item[\rm“user@”, or “user@domain”] \mbox{} \\
Usernames -- domainless (“user@”) or domainfull.

\item[\rm An IP address in {[}nn.nn.nn.nn{]}/prefix form] \mbox{} \\
Unspecified bits must be 0.
(Network IPv6 addresses containing IPv4-mapped addresses are translated
 into plain IPv4.)

\item[\rm A tag -- word begining with underscore] \mbox{} \\
An “alias” dataset entry for “=” “attribute” uses.
\end{description}


{\em attribute} and {\em value} are tokens.
They are used by {\tt policytest()} to make decisions.

The attribute scanners operate in a manner, where the first
instance of interesting attribute is the one that is used.
Thus you can construct setups which set some attribute, and
then {\em ignore} all latter instances of that same attribute
which have been pulled in via “{\em = _alias_tag}” mechanism,
for example.

In following, “understood” value is one or both of literals: “+”, “-”,
if they are listed at the definition entry.
In case only one is understood, the other one can be considered as
placeholder which stops the scanner for that attribute.

Attribute names, and understood value tokens are:

\begin{description}
\item[\tt = _any_tag] \mbox{} \\
The analysis function will descend to look up “_any_tag” from
the database, and expand its content in this place.

\item[\tt rejectnet +] \mbox{} \\
Existence of this attribute pair sets persistent session flag:
“always-reject”, which causes all “MAIL FROM” and “RCPT TO”
commands to fail until end of the session.

This is tested for at the connection start against connecting
IP address, and against IP-reversed domain name.
This is also tested against the “HELO/EHLO” supplied parameter
string.

Use of this should be limited only to addresses against which you
really have grudges.

\item[\tt freezenet +] \mbox{} \\
Existence of this attribute pair sets persistent session flag:
“always-freeze”, which will accept messages in, but all of them
are moved into “freezer” spool directory.

This is tested for at the same time as “rejectnet”.

\item[\tt rejectsource +] \mbox{} \\
Existence of this attribute pair rejects “MAIL FROM” address,
and thus all subsequent “RCPT TO” and “DATA” transactions
until new “MAIL FROM” is supplied.

\item[\tt freezesource +] \mbox{} \\
Existence of this attribute pair causes subsequently following
“DATA” phase message to be placed into “freezer” spool directory.
```

```
This is tested for only at “MAIL FROM”, and subsequent “MAIL FROM”
may supply another value.

\item[\tt relaycustomer +/-] \mbox{} \\
Existence of this attribute pair is tested for at “MAIL FROM”,
and it affects subsequent “RCPT TO” address testing.

Pair “relaycustomer -” is a placeholder no-op, while
“relaycustomer +” tells to the system that it should not
test the “RCPT TO” address very deeply.

{\em Usage of this attribute is not encouraged!
Anybody could get email relayed through just by claiming
a “MAIL FROM” domain which has this attribute.}

\item[\tt relaycustnet +] \mbox{} \\
Existence of this attribute pair is tested for at the  connection
start against connecting IP address, and against IP-reversed domain name.

If this pair exists, session sets persistent “always-accept” flag,
and will not do further policy analysing for the “MAIL FROM”, nor
“RCPT TO” addresses.  (Except looking for valid A/MX data from the
DNS for the sender/recipient domains.)

\item[\tt fulltrustnet +] \mbox{} \\
Because the DNS lookups still done with “relaycustnet +” setting on,
a massive feed for fanout servers might become slowed down/effectively
killed, unless we use “fulltrustnet +” specification for the feeder
host.  Then everything is taken in happily from that source address.

\item[\tt trustrecipients +] \mbox{} \\
This is a variant of “relaycustnet,” where  “RCPT TO” addresses are
not checked at all, but “MAIL FROM” addresses are looked up from
the DNS. (Unless some other test with the “MAIL FROM” domain name
has matched before that.)

\item[\tt trust-whoson +] \mbox{} \\
If the system has been compiled with support to “{\em whoson}” services,
see file “{\em whoson-*.tar.gz}” in the “contrib/” subdirectory.
This facilitates indirectly authenticated (via POP/IMAP) SMTP message
submission for dialup-type users.

\item[\tt relaytarget +] \mbox{} \\
With this attribute pair the current “RCPT TO” address is accepted in
without further trouble. (Theory being, that keys where this attribute
pair exist are fully qualified, and valid, thus no need for DNS analysis.)

See “RCPT TO” processing algorithm for further details.

\item[\tt relaytarget -] \mbox{} \\
This attribute pair causes instant rejection of the current “RCPT TO”
address.

See “RCPT TO” processing algorithm for further details.

\item[\tt freeze +] \mbox{} \\
When “RCPT TO” address test meets this attribute pair, the entire
message will be placed into “freezer” directory.

\item[\tt acceptifmx +/-] \mbox{} \\
This attribute pair is used to give fuzzy feeling in anti-relay setups
so that we don’t need to list {\bf all} those target domains that we
are allowing to use ourselves as relays.

This will basically check that “RCPT TO” address has our server
as one of its MX entries.

The value (“+” or “-”) determines how “severe” the nonexistence
of MX data is.  With “+” the server will yield “400” series temporary
error with implied invitation to try again, and with “-” the server will
yield “500” series permanent error.

\item[\tt acceptifdns +/-] \mbox{} \\
This attribute pair is complementary for the “acceptifmx” in sense
that it accept the recipient address in case the DNS system has any
A or MX information for it.

This attribute pair should not be used.

\item[\tt senderokwithdns +/-] \mbox{} \\
This attribute pair will do DNS analysis for “MAIL FROM” domain, and
accept it only if there exists A or MX data for the said domain.

The value (“+” or “-”) determines how “severe” the nonexistence
of DNS data is.  With “+” the server will yield “400” series temporary
error with implied invitation to try again, and with “-” the server will
yield “500” series permanent error.


\item[\tt sendernorelay +] \mbox{} \\
Tested at “MAIL FROM” address domain, and affects “RCPT TO”
address domain analysis.
{\em At the moment this attribute does not make sense, don’t use!}

\item[\tt test-dns-rbl +] \mbox{} \\
This attribute pair will use Paul Vixie’s RBL
( HTTP://maps.vix.com/rbl/ )
system to block undesired connection sources.

\item[\tt rply-dns-rbl +] \mbox{} \\
\item[\tt test-rply-dns-rbl +] \mbox{} \\
This is a “recipient selective” version of the RBL.
The first one is to be placed into the default address case
(the “[0.0.0.0]/0”), and then the latter can be used in given
destination domain(s) to test for the result of the lookup.

This allows selective usage of ’RBL’ blocking via this server.
For example if you have {\tt smtp-policy.mx} file listing special
cases (opposite of your default domain address “.” values)

\begin{alltt}\medskip\hrule\medskip
 fobar.com  test-rply-dns-rbl + relaytarget +
 barfo.dom  test-rply-dns-rbl + relaytarget +
\medskip\hrule\end{alltt}\medskip

The selectivity can be either by means of listing those where the test
happens, or those where it doesn’t happen -- the latter means that
the default domain address (“.”) must have “test-rply-dns-rbl +” entry.

\item[\tt maxinsize nnn] \mbox{} \\
This attribute pair yields numeric limit for incoming message
size.  With this you can define source specific message size
limits so that if your intranetwork has a system with lower
inbound message size, than you do, you can report this limit
at the “EHLO” responses.

Partly this is placeholder for further code with for example
source/target domain specific runtime enforced size limits.

\item[\tt maxoutsize nnn] \mbox{} \\
Placeholder for further code

\item[\tt localdomain *] \mbox{} \\
Placeholder for further code

\item[\tt message "text string in quotes"] \mbox{} \\
Placeholder for further code
\end{description}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%\end{multicols}
```

17.4.1. Semantics

```
The {\tt policytest()} function is called by smtp-server to check the client
host, the sender’s and recipients’ addresses.
policytest()  looks for  the name and address of the client host
as well as full and partial user address and domain part of sender and
recipient addresses in this database.
The retrieved attributes are used to make decissions on accepting or rejecting
the incoming mail.

If looking for “foo.bar.edu” and an exact match failed, the database
looks for keys in sequence:
  “.foo.bar.edu”, “.bar.edu”, “.edu”, and “.”.

The order of entries in the input file is not important, as the file is
compiled into binary database for faster lookup.

When searching for an IP address the entry with the most common (leftside)
bits is returned.   So you can have a [0.0.0.0]/0 entry what specifies the
default addributes for all unlisted IP addresses. (Both IPv4 and IPv6)

‘=’ is a special attribute.

The notation ‘= _tag’ means “See also at ‘_tag’”.
If server() doesn’t find the requested attribute
of the object, it will replace object name with ‘_tag’ and restart the search.
```

```
{\Large SCRIPT REMOVED; SEE FILE  smtp-policy.src}
```

17.5. Content Based Filtering

The ZMailer **smtpserver** *can* do also message content analysis with an external program at the end of *DATA-dot*-phase, and *BDAT LAST*-phase (that is, when the input message is complete, and final acknowledgement is expected by the email sender.)

The program becomes active if PARAM entry “contentfilter” is set:

```
# External program for received message content analysis:
#PARAM  contentfilter $MAILBIN/smtp-contentfilter
```

The interface to the program is simple synchronous half-duplex one, smtpserver writes relative filepath of the message into programs stdin, ending it with a newline. The filter programs reply must begin with a signed integer, then whatever text is desired to give to the user.

The *contentfilter* program is started without parameters running userid of `daemon` in directory `$POSTOFFICE/`.

The program must silently wait for input, which is full path to the message spool file, analyze it, and reply with exactly one line matching rule of: “`%i `” — begin with signed integer, then have one or more whitespace, then whatever filter writer liked to tell.

General rule:

```
-1 negatives are condemned into rejection
 0 zero is ok! gladly accepted
 1 positives are sent into the freezer
```

The program *may* produce also the numeric SMTP reply codes in its response text:

```
-1
-1 250 2.7.1 Glad to see some spam, immediately destroyed :)
 0
 0 250 2.6.0 Message OK!
 1
 1 550 5.7.1 That is spam, rejected!
```

If the message has no text, some defaults are supplied. If the message text starts with numbers, it is presumed that it contains both the SMTP reply code, and ENHANCEDSTATUSCODE before the text. (If no ENHANCEDSTATUSCODE part is present, then some possibly senseless default is supplied.)

Interface message text lines beginning with anything except signed integer are logged, and the communication channel from the smtpserver to the contentfilter program is closed. Interface continues to scan things reported by the contentfilter program, and if no properly formatted line appears, default is to send the message into the freezer ("-1"); *FIXME! FIXME! "-1" == Kill ??? (Copy&paste from man-page, which may have a bug in it..)*

Chapter 18. Sendmail Reference

This "**sendmail**" program is an emulation of the original sendmail interface. It provides all the original options that make sense to support in the context of ZMailer. This is not intended to be the normal user interface to **mail**, rather it is used by the old User Agent programs, e.g., mail(1), to submit mail. This mechanism has been superseded by the `zmailer(3)` library routines as the native submission interface (Application Program Interface) for ZMailer.

The default action is to submit the RFC822 format mail message expected on `STDIN` to the mailer, with the addresses listed on the command line as recipients. If there are no recipient addresses specified on the command line, the mailer will infer them from the message header. The sender is the account of the current *userid*, except for `root` where the preferred sender is the account of the current login session. The message terminates when a period is seen by itself on a line, or at end of file on the input stream. (modulo used options.)

If the message submission fails immediately on the `mail_open(3)`, the data on `STDIN` will be appended to a `~/dead.letter` file in the submitters home directory.

Usage:

```
/usr/sbin/sendmail: unknown option -?
Usage: sendmail [sendmail options] [recipient addresses]
 ZMailer's sendmail recognizes and implements following options:
   -B bodytype   -  Valid values: 8BITMIME, 7BIT
   -C conffile   -  specifies config file (meaningfull for -bt)
   -E            -  flag 'external' source
   -F 'full name'   sender's full name string
   -N notifyopt -  Notify option(s): NEVER or a set of:
                                     SUCCESS,DELAY,FAILURE
   -P priority# -  numeric priority for ZMailer router queue
                    pre-selection
   -R returnopt -  Error report return option, either of: FULL, HDRS
   -U            -  Flag as 'user submission'
   -V envidstring - XTEXT encoded ENVID string
   -b?           -  operational mode flags
   -bd           -  starts smtpserver in daemon mode
   -bi           -  runs 'newaliases' command
   -bm           -  deliver mail; always :-)
   -bp           -  runs 'mailq' command
   -bs           -  speak smtp; runs server in interactive mode
   -bt           -  starts router in interactive test mode
   -e*           -  (ignored)
   -f fromaddr   -  sets envelope from address for the message
   -i            -  on inputs from tty this will ignore SMTP-like
                    dot-EOF
   -m            -  send a copy of the message to the sender too
                    (ignored)
   -o*           -  multiple options; those not listed cause error
   -oQ queuedir -  defines POSTOFFICE directory for message
                    submission
   -ob*          -  (ignored)
   -od*          -  (ignored)
   -oe*          -  (ignored)
   -oi           -  alias of '-i' option
   -or*          -  (ignored)
```

```
-p submitprotocol - value for 'with' label at 'Received:' header
-q*          -  queue processing commands (ignored)
-r fromaddr  -  (alternate for -f)
-t           -  scan message rfc822 headers for recipients
-v           -  verbose trace of processing
```

Parameters:

`-bm`

asks **sendmail** to deliver mail, which it does anyway. This option has no effect.

`-bs`

will start a SMTP server reading from `STDIN`. This causes the **smtpserver(8)** program to be executed.

`-bd`

starts the **router(8)** and **scheduler(8)** programs to emulate sendmail's daemon mode. This is *not* a recommended method to start these programs, instead use **zmailer(1)** script.

`-bt`

runs the **router(8)** in interactive mode for testing.

`-bu`

runs **newaliases(8)** to rebuild the alias file database.

`-bp`

runs **mailq(1)** to print the mail transport queue status.

`-C configfile`

specifies the **router(8)** configuration file.

`-E`

indicates the origin of this message is an insecure channel. This should be used when **sendmail** is used to submit messages coming in from outside the local machine, to avoid security problems during message processing. This flag ensures the message will have no privileges even if the current userid is "trusted".

`-f address`

specifies the sender address. This is the default originator address if there is no "From:" headerin the message. It becomes the "Sender:" address otherwise. In either case if the current userid is not "trusted" by the mailer, it is free to ignore both this option and any header information to ensure properly authenticated originator information.

`-F fullname`

specifies the full name of the (local) sender.

`-i`

tells **sendmail** to not use a period (".") on a line by itself as a message terminator, only the end of file will terminate the message.

`-m`

asks the mailer not to ignore the originator in the addressee list. This is default behaviour, so this option has no effect.

`-N notify`

sets Delivery-Status-Notification notify parameter to be: `NEVER`, or any combination of: `SUCCESS`, `FAILURE`, `DELAY`.

`-oi`

is like `-i`.

`-oQ postoffice`

specifies an alternate `$POSTOFFICE/` directory.

`-q`

asks for queue processing. This option has no effect.

`-Q retmode`

sets Delivery-Status-Notification parameter to be either of: `FULL`, `HDRS`.

`-r address`

is like `-f`.

`-t`

scans header for recipient addresses if none are specified on the command line. This is also the default behaviour, so this option has no effect.

`-v`

will report the progress of the message after it has been submitted. The **sendmail** process will write verbose log information to the `STDERR` stream until the **scheduler** deletes the message.

`-V envid`

sets Delivery-Status-Notification parameter `ENVID` to be any arbitrary *[xtext]* string.

Chapter 19. Rmail Reference

The **rmail** is a program to process incoming UUCP mail. **rmail** is usually invoked by a remote UUCP neighbour host's mailer using a command line like:

```
uux - -r -asender -gC thishost!rmail (recipient1) (recipient2) ...
```

The end result is that the remote neighbour's **uuxqt**(8) runs **rmail** on **thishost** with this command line:

```
rmail recipient1 recipient2 ...
```

In both cases, a UUCP format mail message is on the standard input.

The task of **rmail** is to transform the trace information in the UUCP format message to the equivalent RFC822 trace information, and to submit the message to the **zmailer**(1) router with the appropriate envelope information.

The expected input format looks like:

```
  From address3  date3 remote from host3
  >From address2  date2 remote from host2
  >From address1  date1 remote from host1
```

followed by the rest of the message. This is considered equivalent to the following (as it might appear in a mailbox):

```
  From host3!host2!host1!address1 date
  Received: by host3 ... ; date3
  Received: by host2 ... ; date2
  Received: by host1 ... ; date1
```

In order for the mailer to process the incoming message properly, **rmail** must be run by a "*userid*" which the **router**(1) will accept forged mail from. his is normally the UUCP account id.

Usage:

rmail [-d] [-h *somewhere*] *recipient*...

Parameters:

`-d`

turns on debugging output.

`-h` somewhere

will use the argument as the default remote UUCP host name to use if there is no *remote from host* tag in the first *From-space* line in the message. The default value for this is usually *somewhere* or *uunet* (since uunet was a frequent purveyor of this protocol violation).

Chapter 20. zmailer(3) Reference

Usage:

```
    #include <stdio.h>
#include <zmailer.h>

FILE *mail_open(char *);

int mail_priority;

int mail_abort(FILE *);

int mail_close(FILE *);

int mail_close_alternate(FILE *mfp, char *where, char *suffix);

char *mail_alloc(unsigned int);

int mail_free(char *);

char *mail_host();
```

At linkage time use **`-lzmailer`**.

`mail_open()` will return a `FILE *` to a message file that should be written to by the application. This message file contains three parts:

- the message envelope
- the message header
- the message body

The exact format of these components depend on the message protocol, which must be specified as the parameter to `mail_open()`. The choices are predetermined by the capabilities of the mailer, and are defined in the header file. The known possibilities are:

`MSG_RFC822`

: this is the only format supported by default by the mailer. The message headers and body in this format are defined by the IETF Request For Comments 822 and 1123. The message envelope syntax is similar to the message header syntax.

`MSG_FAX`

: intended for fax transmissions. (never used)

`MSG_UUCP`

: intended for old style UUCP format message headers (never used)

`MSG_X400`

: intended for X.400(88) messages.

The `mail_open()` routine will look for $`FULLNAME` and $`PRETTYLOGIN` environment variables and translate them into message envelope data for use by the mailer if it generates a sender address header for the message.

Note that the return value from the `mail_open()` routine corresponds to the return value of an `fopen(3)`, and similarly the return values from `mail_abort()` and `mail_close()` correspond to the return value of `fclose(3)`.

The `mail_priority` variable has a default value of 0, and is used on scanning Zmailer configuration variable $`ROUTERDIRS`, which tells alternate router directories under the `$POSTOFFICE/` directory. At value 0, $`ROUTERDIRS` variable is not used. At higher values, successive directory from $`ROUTERDIRS` is taken. See below about Z-Environment.

The `mail_close_alternate(3)` can be used to send currently open message file to some alternate destination, and is used at *smtpserver(8)* to send some quick-action requests directly to the *scheduler(8)*.

The `mail_alloc()` and `mail_free()` routines are used to provide memory space for internal data structures. The versions of these routines in the library simply call `malloc(3)` and `free(3)` but an application may override them if desired.

Similarly the `mail_host()` routine is intended to return a unique string for each host, by default the hostname, and this too is intended to be overridden by an application that may already have this information available in some form.

Envelope header lines:

The message envelope headers are used to carry meta-information about the message. The goal is to carry transport-envelope information separate from message (RFC-822) headers, and body. At first the message starts with a set of envelope headers (*-prefix denotes optional):

```
 *external \n
 *rcvdfrom %s@%s (%s) \n
 *bodytype %s \n
 *with %s \n
 *identinfo %s \n

Either:
  from <%s> \n
Or:
  channel error \n

 *envid %s \n
 *notaryret %s \n

Then for each recipient pairs of:

 *todsn [NOTIFY=...] [ORCPT=...] \n
 to <%s> \n

Just before the data starts, a magic entry:

 env-end \n
```

Then starts the message RFC-822 headers, and below it, the body.

Example:

```
... set up signal handlers ...
FILE *mfp = mail_open(MSG_RFC822,0);
if (mfp != NULL) {
     ... output the mail message to mfp ...
} else
     ... error handling for not being able to open the file ...
if (some application processing went wrong
          || we took an interrupt)
     (void) mail_abort(mfp);
else if (mail_close(mfp) == EOF)
     ... error handling if something went wrong ...
```

Environment variables:

`FULLNAME`

variable defines textual fullname, for example: “Sample User”

`PRETTYLOGIN`

variable defines *user@node* format of what user wants to claim as his/her own address (it must match those of mail router accepts.)

Z-environment variables:

`POSTOFFICE`

defines the directory where all `$POSTOFFICE/` functions are. Example:

```
POSTOFFICE=/var/spool/postoffice/
```

`ROUTERDIRS`

defines a “:” separated list of alternate router directories. If these are defined at all, they must exist, if alternate queueing priority mechanism is desired to be used. Example:

```
ROUTERDIRS=router1:router2:router3:router4
```

Chapter 21. Router Reference

The **router** daemon makes all decisions affecting the processing of messages in ZMailer.

A mail message is submitted by placing it in a file in the `$POSTOFFICE/router/` directory. The **router** frequently scans this directory for new files and will lock and process them as it finds them. The result is a message control file that gets linked into the `$POSTOFFICE/scheduler/` and `$POSTOFFICE/transport/` directories for use by the **scheduler** in the next step of message processing. The original message file is then moved to the `$POSTOFFICE/queue/` directory.

The **router**'s behaviour is controlled by a configuration file read at startup. It is really a **zmsh**(1) script that uses facilities provided builtin to the **router**.

Usage:

Invoking router without any arguments will do nothing (except make it read its configuration file and promptly exit). The normal startup method is to run the **zmailer**(1) script, as in "**zmailer router**". This will start the **router** as a daemon and kill the previous incarnation of the **router**.

router [-diksSV] [-f configfile] [-n #routers] [-o zmsh-options] [-t traceflag] [-L logfile] [-P postoffice]

Parameters:

`-d`

Detach and run as a daemon.

`-f` *`configfile`*

Overrides the default configuration file `$MAILSHARE/router.cf`.

`-i`

Run interactively, presenting a **zmsh** session with the configuration file preloaded.

`-k`

Kill the currently running router by sending it a `SIGTERM` signal.

`-L` *`logfile`*

Overrides the default log file location `$LOGDIR/router`.

`-n` *`#routers`*

Starts the specified number of parallel router processes. The default is a single router process.

`-o` *`zmsh-options`*

Sets the option string passed on the internal **zmsh** invocation. The default is `-O`. Note that the leading "-" is mandatory. See ***zmsh(1)*** (at Section 21.1.2) for the available options.

`-P` *`postoffice`*

Specifies an alternate `$POSTOFFICE/` directory.

`-S`

Can be used to turn off non-serious syslogging.

`-s`

Turns stability-flag off and on. Without this flag, the search of new jobs will be done with (sometimes) timeconsuming care of organizing the job files into time order.

`-t` *`traceflag`*

Sets trace options, one per `-t` switch, even before the configuration file is loaded. This is otherwise equivalent to the builtin **trace** command. The currently known options are: `assign, bind, compare, db, final, functions, matched, memory, on, regexp, resolv, rewrite, router, and sequencer.`

`-V`

Print version message and run interactively.

To restart a **router** daemon:

```
router -dk
```

To test an address, start up an interactive session:

```
router -i
```

or if the ZMailer **sendmail**(8) is installed:

```
sendmail -bt
```

Then just use the pre-defined functions.

21.1. ZMSH Script Language

zmsh is an implementation of the Bourne shell suitable for use with the ZMailer **router**(8) as its configuration file language interpreter. It contains extensions that allow structured data (in the form of lists) to be manipulated.

The shell supports three basic kinds of functions: Unix commands, user-defined functions, and builtin commands. The latter comes in two variations: normal functions which take string arguments and return a status code (much as an embedded Unix command would work), and list-manipulation functions which understand list arguments and can return list arguments. The defined functions can take any form of argument and return any form of value (a status code, a string, or a list).

Shell operations (pipes, backquote evaluation and substitution) will work between combinations of builtin functions, defined functions, and Unix commands.

The shell precompiles its input to a (possibly optimized) byte-code form, which is then interpreted as required. This means that the original form of the input is not kept around *in-core* for future reference. If the input is an included file, the shell will try to save the byte-code form in a `.fc` file associated with the input file. For example, if input is from `file.cf`, the shell will try to create `fc/file.fc` and then `file.fc`. These files will in turn be searched for and loaded (after a consistency check) whenever a `.cf` file is included.

The effects of input and output redirections are predicted prior to the execution of a command and its I/O setup.

21.1.1. ZMSH Usage:

zmsh [`-CILOPRSYaefhinstuvx`] [`-c` *`command`*] [*`script ...`*]

21.1.2. ZMSH Parameters:

`-c command`

Run the given argument as a shell command script.

`-a`

Automatically export new or changed shell variables.

`-e`

Exit on non-zero status return of any command.

`-f`

Disables filename generation.

`-h`

Hash and cache the location of Unix commands. The option is set by default.

`-i`

This shell is interactive, meaning prompts are printed when ready for more input, `SIGTERM` signal is ignored, and the shell does not exit easily. This flag is automatically set if `stdin` and `stderr` are both attached to a `/dev/tty`.

`-n`

Read commands but do not execute them.

`-s`

Read commands from `stdin`. If there are non-option arguments to the shell, the first of these will be interpreted as a shell script to open on `stdin`, and the rest as arguments to the script.

`-t`

Exit after running one command.

`-u`

Unset variables produce an error on substitution.

`-v`

Print shell input as it is read.

`-x`

Print commands as they are executed.

21.1.3. ZMSH Debug options:

`-C`

Print code generation output onto `stdout`. If this option is doubled, the non-optimized code is printed out instead.

`-I`

Print runtime interpreter activity onto `/dev/tty`.

`-L`

Print lexer output onto `stdout`.

`-O`

Optimize the compiled script. If this option is doubled, the optimized code is also printed out.

`-P`

Print parser output (S/SL trace output) onto `stdout`.

`-R`

Print I/O actions onto `/dev/tty`.

`-S`

Print scanner output (token assembly) onto `stdout`.

`-Y`

Open `/dev/tty` for internal debugging use.

21.2. Configuration Script Writing Rules

```
Text to be inserted here.
```

21.3. Script Security Issues

```
Text to be inserted here.
```

21.4. Router Script Well Known Entrypoints

This section describes the **router** internal functions used as entrypoints by various uses inside and outside the **router** program.

21.4.1. The `process()` function

FIXME! WRITEME!

21.4.2. The `router()` function

FIXME! WRITEME!

21.4.3. The `crossbar()` function

FIXME! WRITEME!

21.4.4. The `server()` function

FIXME! WRITEME!

21.5. Script Language Internal Functions

This section describes the **router** internal functions.

FIXME! FIXME! some internal functions are missing from this listing!

21.5.1. "`:`" (doublecolon)

Syntax:

: <SPC> *`anything else forming a command pipeline`*

Description:

none

Return Values:

0

Options:

none

Notes:

none

21.5.2. "`.`", "`include`"

Syntax:

```
. scriptfilename
```

Alternate syntax:

include *`scriptfilename`*

Return Values:

Exit status of script evaluation, or specifically:

1

File not found, or not fstat:able.

2

Internal loadeval() didn't yield same result as fstat:ed file size is.

64

Usage: Not exactly one parameter, or it is a void string.

Options:

none

Notes:

This puts the running script to read more script from given filename.

21.5.3. "`[`", "`test`"

Syntax:

[*`test parameters`* **]**

Alternate syntax:

test *`test parameters`*

Return Values:

1

True.

0

False.

-1

Error.

Options:

- File testing unary prefix functions:

  `-b file`

  True if file exists and is block special.

  `-c file`

  True if file exists and is character special.

  `-d file`

  True if file exists and is a directory.

  `-f file`

  True if file exists and is a regular file.

  `-g file`

  True if file exists and is set-group-id.

  `-k file`

  True if file has its "sticky" bit set.

`-p file`

True if file exists and is a named pipe.

`-r file`

True if file exists and is readable.

`-s file`

True if file exists and has a size greater than zero.

`-t [fd]`

True if `fd` is opened on a terminal. If `fd` is omitted, it defaults to 1 (standard output).

`-u file`

True if file exists and its set-user-id bit is set.

`-w file`

True if file exists and is writable.

`-x file`

True if file exists and is executable.

- String testing binary functions:

`str1 = str2`

True if the strings are equal.

The "str1" must not begin with a hyphen "-" character! (Actually they must not be any of the magic prefix-, or infix keyword operators, nor "(", "!", or ")".)

Suggested test setup:

```
[ "x$varname" = "xliteral" ] ...
```

`str1 != str2`

True if the strings are not equal.

The "str1" must not begin with a hyphen "-" character! (Actually they must not be any of the magic prefix-, or infix keyword operators, nor "(", "!", or ")".)

Suggested test setup:

```
[ "x$varname" != "xliteral" ] ...
```

- Integer value testing binary functions:

`iexpr -eq iexpr`

True if integer values are equal.

The iexpr intermediate products are strings for the purposes of this scanners input, but then they are internally treated as system local "integer" datatype.

The *second* iexpr can be a pair of: "-l string-expr" which is evaluated as *length* of that string-expr.

`iexpr -ne iexpr`

True if integer values are not equal.

The *second* iexpr can be a pair of: "-l string-expr" which is evaluated as *length* of that string-expr.

`iexpr -gt iexpr`

True if integer value1 is greater than integer value2.

The *second* iexpr can be a pair of: "-l string-expr" which is evaluated as *length* of that string-expr.

`iexpr -ge iexpr`

True if integer value1 is greater or equal than integer value2.

The *second* iexpr can be a pair of: "-l string-expr" which is evaluated as *length* of that string-expr.

`iexpr -lt iexpr`

True if integer value1 is less than integer value2.

The *second* iexpr can be a pair of: "-l string-expr" which is evaluated as *length* of that string-expr.

`iexpr -le iexpr`

True if integer value1 is less or equal than integer value2.

The *second* iexpr can be a pair of: "-l string-expr" which is evaluated as *length* of that string-expr.

- File comparison binary functions:

`file1 -nt file2`

True if file1's mtime is newer than file2's.

Filenames are best to be either absolute paths (begins with "/"), or dot-relative (begins with "./" pair). Unqualified names are hazardous, e.g. "-file-name-".

`file1 -ot file2`

True if file1's mtime is older than file2's. (See comment above.)

`file1 -ef file2`

True if both files have same inode, and device. (See comment above.)

- Logical functions:

`! expr`

Unary NOT

`expr -a expr`

Binary AND

`expr -o expr`

Binary OR

`( expr )`

Parenthesis

Notes:

This is basically the shell "`[`" a.k.a. "`test`" program.

Do note that unlike more usual bourne-shells, this *does not* short-circuit logical evaluations, e.g. falseness of the left side of an AND does not eliminate evaluation of the right side!

Usual "`test`" precautions with things like parameter data prefixing with some character so that it will not become treated as one of the control options.

```
Not:
  [ "$(funcname ...)" ]
  [ "$varname" = "value" ]

Ok:
  [ "z$(funcname ...)" ]
  [ "z$varname" = "zvalue" ]
```

21.5.4. `attributes`

Syntax:

attributes *`object-reference`*

Return Values:

The property list symbol (4th) component of an address quad.

Options:

none

Notes:

none

21.5.5. `basename`

Syntax:

basename *`pathname`* [*`suffix`*]

Return Values:

0
: ok, result string to `stdout`.

1
: Error.

Options:

If a suffix is given and matches the filename, the suffix too is stripped from the filename.

There are no "-"-type options.

Notes:

Prints the base filename of the pathname.

21.5.6. `break`

Syntax:

```
while ...
do
   ...
   break
   ...
done

case ...
   ...
   break
   ...
esac
```

Return Values:

1

Options:

none

Notes:

none

21.5.7. `builtin`

Syntax:

NOT IMPLEMENTED!

builtin {*`builtin command name, and its params`*}

Return Values:

0

Options:

none

Notes:

NOT IMPLEMENTED!

Ensures that named command is builtin within the scheduler, and not a) external, b) script based.

21.5.8. `car`

Syntax:

car *`object-reference`*

Return Values:

Pointer to copied lisp-object of object-reference's `car` operation.

Options:

none

Notes:

This command is synonymous to the *`channel`* (at Section 21.5.12), and *`first`* (at Section 21.5.25) functions.

This version name is for die-hard LISP fans ;-)

21.5.9. `cat`

Syntax:

cat *`filenames`*...

Return Values:

Output into `stdout`, which can be piped ZMailer router script internally to next receiver in a pipeline.

Options:

none

Notes:

The filenames to be "cat:ed" together need to be regular files.

21.5.10. `cd`

Syntax:

cd [*`directory`*]

Return Values:

0

`cd` successful.

1

Error, can't find `$HOME` environment variable.

1

Error, internal `chdir`(2) call failed. The `stderr` gets an error string.

64

Bad parameters (too many). (Usage.)

Options:

none

Notes:

none

21.5.11. `cdr`

Syntax:

cdr *`object-reference`*

Return Values:

Pointer to copied lisp-object of object-reference's `cdr` operation.

Options:

none

Notes:

This command is synonymous to the *`rest`* function (at Section 21.5.51).

This version name is for die-hard LISP fans ;-)

21.5.12. `channel`

Syntax:

channel *`object-reference`*

Return Values:

The channel (1st) component of an address quad.

Options:

none

Notes:

The *`car`* (at Section 21.5.8), and *`first`* (at Section 21.5.25) functions are synonymous to this one.

21.5.13. `continue`

Syntax:

continue

Return Values:

1

Options:

none

Notes:

This affects control flow by returning it into beginning of enclosing loop construction.

Usage examples:

```
while ...
do
   if ... something ...
   then
     ... something too ...
     continue
   fi
   ... more something ...
done
```

21.5.14. `daemon`

Syntax:

daemon

Return Values:

0

Options:

none

Notes:

Starts the **router** running in daemon mode, scanning the `$POSTOFFICE/router/` directory every few seconds for message files to process. This function is invoked automatically by other code in the **router** program and has no other purpose.

21.5.15. `db`

Syntax:

db add *`database key value`*
db flush *`database`*
db owner *`database`*
db print *`database`*
db remove *`database key`*
db count *`database`*
db toc

Return Values:

0

1

Error.

Options:

`a[dd]`

Add a *`key,value`* entry to the database, if possible.

`f[lush]`

For "incore" database this means deletion of the content, but for others this is IO buffer flush (in case of modifications.)

`o[wner]`

Print the account name of the owner of the database, if possible. This is usually determined by the files associated with the database.

`p[rint]`

Print all entries of the database, if possible.

`c[ount]`

Iterate thru the database and count entries in there.

`r[emove]`

Remove a *`key`* entry from the database, if possible.

`t[oc]`

Print a table of defined relations and their associated information. This table has five columns, in order:

- the name of the relation
- its type and subtype
- cache entries and maximum cache size
- flags
- and associated files

See the `relation` function for more information. (Section 21.5.50)

Notes:

DB type	add	flush	owner	print	remove	count
incore	yes	yes	yes	yes	yes	yes
header	yes	yes	yes	yes	yes	yes
ordered	no	yes	yes	yes	no	yes
unordered	yes	yes	yes	yes	yes	yes
hostsfile	no	no	no	yes	no	no
bind	no	no	no	no	no	no
ndbm	yes	yes	yes	yes	yes	yes
gdbm	yes	yes	yes	yes	yes	yes
dbm	yes	yes	yes	yes	yes	yes
yp (NIS)	no	no	yes	yes	yes	no
bhash	yes	yes	yes	yes	yes	yes
btree	yes	yes	yes	yes	yes	yes
ldap	no	no	yes	no	no	no
selfmatch	no	no	no	yes	no	yes

21.5.16. *dblookup*

Syntax:

Does lookup via `relation` defined *dblookup* vector:

dblookup *`key`* [[`-:` *`defaultkey`*] `--` %subst1 %subst2 ... %subst9]

dblookup *`key`* [[`-:` *`defaultkey`*] `--` %subst2 %subst2 ... %subst9]

The latter form is for databases which have *any* driver routine in use!

Return Values:

cell

Lookup result

`NULL`

Lookup failed, variable `$defer` may be set if the reason is considered *temporary* and thus solvable latter.

Options:

The ’-:’ does supply the "final case" lookup key in cases where various domain shortening lookups reach their end (and have the builtin ultimate default lookup of ".").

```
relation -lm%:t $DBTYPE -f $MAILVAR/db/routes$DBEXT routesdb
 ...
a=$(routesdb $lookupkey -: .:ERROR -- $subst1 $subst2)
```

With those things the "routes" database can now contain data like:

```
.:ERROR  smtp!
cust.1   smtp!%0
cust.2   smtp!%0!%1
```

With "subst*" values below, results would be:

```
subst1="foo1"
subst2="foo2"

smtp!
smtp!foo1
smtp!foo1!foo2
```

respectively.

Note: The %subst things are used only if the relation definition has -% option flag set!

Notes:

The access function to the database facilities in the **router**.

FIXME! Notes about %[0-9] substitution rules, and their controls. (That is: `relation`*’s* `-%` *option! Section 21.5.50)*

A complete example of "`-:`" usage from `p-routes.cf`:

```
routes_neighbour (domain, address, A) {
    local tmp

    # We have Alternate default-lookup for cases of locally generated
    # ERROR MESSAGES -- for case where the dot (.) leads to error!
    # and we want to reply with DIFFERENT address, thus:  '.:ERROR' key.

    $(iserrormsg) &&
        tmp=$(routesdb "$domain" -: '.:ERROR') &&
            tmp=$(routes_spec "$tmp" "$address" $A) &&
                returns $tmp

    tmp=$(routesdb "$domain") &&
        tmp=$(routes_spec "$tmp" "$address" $A) &&
            returns $tmp

#| The routes_spec function interprets the return value from the
#| routesdb lookup.

    return 1
}
```

21.5.17. `echo`

Syntax:

echo [`-n`] [`--` *`string-expressions`*...]

Return Values:

0

Options:

`-n`

Don't print newline at end of string.

`--`

End the option set, use this if there is even a small change of starting the the data with the hyphen character.

Notes:

none

21.5.18. `elements`

Syntax:

elements *`lisp-object`*

Return Values:

lisp-object

Options:

none

Notes:

This does a side-effect on input list, which is need in "for" loops:

```
for loopvar in $(elements $listvar)
do
  ...
done
```

21.5.19. `envars`

Syntax:

envars

Return Values:

0

Options:

none

Notes:

A debug tool to print internal variable tree.

21.5.20. `erraddron`

Syntax:

erraddron [*`file`*]

Return Values:

0

Successfull operation.

64

Wrong usage of arguments.

Options:

Optional filename.

Notes:

Without a filename this dissociates the possible pre-existing logging file definition.

With a filename option this specifies a filename into which the router appends all address parsing error messages. This is primarily for curious postmasters or other collectors of address trivia.

This is a debug tool; usage examples:

```
erraddron $POSTOFFICE/postman/ERRADDRLOG

erraddron
```

21.5.21. `eval`

Syntax:

eval *`expression`*

Return Values:

status

Options:

none

Notes:

This is the generic workhorse for self-modifying code execution within the *zmsh*, it creates the workhorse of following code-fragment:

```
# Usage: newattribute <oldattribute> <key1> <value1> [ <key2> <value2> ] ...
#
# Returns a new attribute list symbol with the <keyN> <valueN>
# attributes added to the contents of the <oldattribute> list.

newattribute (oldattribute) {
        local a null value

        a=$(gensym)
        eval $a=\$$oldattribute
        while [ "$#" != 0 ];
        do
                lreplace $a "$1" "$2"
                shift ; shift
        done
        echo -- "$a"
}
```

21.5.22. `exit`

Syntax:

exit

Return Values:

does not return, does `exit(2)` for the shell/router..

Options:

none

Notes:

Exit from the shell/router with exit code.

21.5.23. `export`

Syntax:

export [*`variable-name`*...]

Return Values:

0

??? FIXME

Options:

none

Notes:

Exports variable name(s). If no variables are given, export prints a list of which variables have been exported.

21.5.24. `filepriv`

Syntax:

filepriv *`file`* [*`uid`*]

Return Values:

0
: ??? FIXME

1
: Error.

Options:

none

Notes:

Prints the numeric user id of the least privileged account that can modify the specified file.

This is determined by an approximation that pessimistically assumes that any file or directory writable by group or others is insecure, and optimistically assumes that it is enough to check a file and its parent directory instead of all the way to the filesystem root. The reason for the latter is that if grandparent directories are insecure, the system is likely to have just as bad potential problems as can be created by using mail to run processes with forged powers (besides, doing the full check would be quite expensive).

If a second argument is given, it is the numeric user id to assume for the file. This means only the parent directory will be checked for nonwritability and for having the same (or a 0) uid.

21.5.25. `first`

Syntax:

first *`object-reference`*

Return Values:

Pointer to copied lisp-object of object-reference's `car` operation.

Options:

none

Notes:

This command is synonymous to the *car* (at Section 21.5.8), and *channel* (at Section 21.5.12) commands.

21.5.26. gensym

Syntax:

gensym

Return Values:

0

Options:

none

Notes:

Generates and prints a new symbol name in the sequence `g0` to `gN` every time it is called. The sequence is reset and any symbol values destroyed after the **router** has processed a message. This function is used to generate new symbols, to hold attached address property lists, during alias expansion.

Code-fragment showing a way how it is used inside the router scripts:

```
# Usage: newattribute <oldattribute> <key1> <value1> [ <key2> <value2> ] ...
#
# Returns a new attribute list symbol with the <keyN> <valueN>
# attributes added to the contents of the <oldattribute> list.

newattribute (oldattribute) {
        local a null value

        a=$(gensym)
        eval $a=\$$oldattribute
        while [ "$#" != 0 ];
        do
                lreplace $a "$1" "$2"
                shift ; shift
        done
        echo -- "$a"
}
```

21.5.27. `get`

Syntax:

get property-list keyname
get property-list-varname keyname

Return Values:

property-list

Options:

none

Notes:

Returns a property-list corresponding to key string:

```
listvar=(key1 value1 keyname value key3 value3)
result1=$(get  listvar keyname)
result2=$(get $listvar keyname)
```

21.5.28. `getopts`

Syntax:

getopts *`optstring-name`* [arguments...]

Return Values:

0

success ?

1

Error.

Options:

none

Notes:

Never used inside the *zmsh* scripts, usefullness questionable, and usage documentation missing.

21.5.29. `grind`

Syntax:

grind *`lisp-object`*

Return Values:

Pointer to list of *varcell*.

Options:

none

Notes:

This is a debug tool.

The side-effect is to output the text form of the *lisp-object*.

21.5.30. `groupmembers`

Syntax:

groupmembers *`groupname`*

Return Values:

0

Found data, side-effect is described below.

1

Didn't find anything. No side-effects.

64

Usage (number of arguments) is wrong.

Options:

none

Notes:

Prints the accounts that are listed as members of a group in the system groups file, one per line. Note that accounts with the same login group id, but that are not listed in the groups file, will not appear in this list.

21.5.31. `hash`

Syntax:

hash [`-r`] [*`command names`*]

Return Values:

0

This value is returned always.

Options:

`-r`

Path is flushed.

`command names`

These are looked up one by one via the PATH environment variable, and results are stored into internal quick-access hash table.

Notes:

This is a part of the "*zmsh*" shell-script interpreter in good SH-tradition, but is not used in the ZMailer *router* in any way.

21.5.32. `homedirectory`

Syntax:

homedirectory *`user`*

Return Values:

0

Found home directory, side-effect is printing of that directory into `stdout`, whence it may be captured into e.g. some variable.

2

Error, no such user.

3

Error, some temporary lookup failure. Also sets "$defer" variable.

64

Usage error, wrong number of arguments. (Usage.)

Options:

none

Notes:

Prints the home directory of the specified user account.

21.5.33. `host`

Syntax:

host *`object-reference`*

Return Values:

The host (2nd) component of an address quad.

Options:

none

Notes:

none

21.5.34. `hostname`

Syntax:

hostname [*`name`*]

Return Values:

0

Returns always this value.

Options:

none

Notes:

> Sets the *router*'s idea of the system hostname. Without an argument the name is retrieved from the system and printed. The *router* has no preconceived notion of what the hostname is, so "Message-Id:" and "Received:' headers will only be generated if a "hostname" has been set using this function.

21.5.35. `ifssplit`

Syntax:

ifssplit `any-string`

Return Values:

> split list

Options:

> none

Notes:

> This splits given input string per IFS environment variable, and produces a list of substrings ready for further use.

21.5.36. `lappend`

Syntax:

lappend `varname anyvalue`

Return Values:

> NULL; actually error cases output to `stderr`, all others are handled *silently*.

Options:

> none

Notes:

> This appends content of list object (*anyvalue*) to named *varname* variable:

```
#
# From  aliases.cf  of system standard scripts.
#

maprrouter (attribute, localpart, origaddr, plustail, domain) {
    local shh al
```

```
        al=()
        while read address
        do
            case "$address" in
            ") shh=(((error expansion "$localpart")))
                lappend al $shh
                continue
                ;;
            esac

            defer="
            shh=$(rrouter "$address" "$origaddr" $attribute \
                 "$plustail" "$domain")
            [ -n "$defer" ] &&
                shh=(((hold "$defer" "$address" $attribute)))
            defer="
            lappend al $shh
        done
        returns $al
    }
```

21.5.37. `last`

Syntax:

last *`lisp-object`*

Return Values:

Return a pointer to last cell of *varcell*'s list.

Options:

none

Notes:

This is actually a unused relict from way way back..

21.5.38. `length`

Syntax:

length *`lisp-object`*

Return Values:

String of decimal numbers representing the number of *varcell*'s in the lisp-object primary chain.

Options:

none

Notes:

This counts the number of elements in the chain, not length of any string.

21.5.39. `list`

Syntax:

list *`objects`*...

Return Values:

List-wrapped lisp-object.

Options:

none

Notes:

```
z# list 1 2 3
(1 2 3)
z# tt=(1 2 3)
z# list $tt
((1 2 3))
z# tt=$(list 1 2 3)
z# grind $tt
(1 2 3)
```

21.5.40. `listaddresses`

Syntax:

listaddresses [`-e` *`error-address`*] [`-E` *`errors-to-address`*] [`-c` *`comment`*]

The `stdin` feeds in the list of addresses.

Return Values:

lisp-list

Successfull processing result. There can still be an error-report sent to the error-address; see below.

NULL

Error, `stderr` gets an error report text, and in some cases also the "`-e`" defined error-address, and "postmaster" will get email telling of problems in the file.

Options:

`-e`

Any syntax errors at list parsing will cause a report to be mailed to the given address.

`-E`

If an error occurs while messages are being delivered, the 'errors-to-address' can be used to force error message destination elsewhere than to the default 'sender' of the message.

`-c`

A comment will be inserted in the error report.

Notes:

Filters an RFC822 address list on standard input to produce one normal form (no non-address tokens) address per line on its output. This function can be used to parse the alias file or .forward files or similar.

21.5.41. `listexpand`

Syntax:

listexpand [`-c` *`comment`*] [`-e` *`error-address`*] [`-E` *`errors-to-address`*] [`-p` *`privilege-integer`*] [`-N` *`notary-string`*] *`$attribute`* *`$localpart`* *`$origaddr`* [*`$plustail`* [*`$domain`*]]

Return Values:

lisp-list (or NULL)

Options:

`-e`

Any syntax errors at list parsing will cause a report to be mailed to the given address.

`-E`

If an error occurs while messages are being delivered, the "errors-to-address" can be used to force error message destination elsewhere than to the default "sender" of the message.

`-c`

A comment will be inserted in the error report.

`-p`

Integer privilege code for expanded addresses

`-N`

Notary string data, or "-" for "no DNS".

Notes:

This implements the most common pipeline where *`listaddresses`* (at Section 21.5.40) was used with more efficient memory consumption handling. (System memory usage internals have changed over the time, and now this is no longer especially great memory expenditure saver.)

The `stdin` will feed addresses from a file for parsing, and parameter mapping + routing.

Comparison of previous *`listaddresses`* (at Section 21.5.40) -script implemented code, and new one using the `listexpand`:

Old:

```
l="$preowner$(basename "$lcuser" -mod)$postowner"
nattr=$(newattribute $attr privilege $priv sender "$l")
$(zapDSNnotify $nattr delivered "$sender" "$lcuser$domain")
a=$(runas $priv cat "$a" | \
    listaddresses       -E "$l$domain"          \
                        -e "$l"                 \
                        -c "$a file expansion" |
    maprrouter $nattr "$a" "$host" "$plustail"  \
               "$domain")
postzapDSNnotify a
returns $a
```

New:

```
l="$preowner$(basename "$lcuser" -mod)$postowner"
if priv=$(getpriv "664" $priv "$a" maillist) &&
   nattr=$(newattribute $attr privilege $priv sender "$l") ; then
    $(zapDSNnotify $nattr delivered "$sender" "$lcuser$domain")
    a=$(runas $priv cat "$a" | \
        listexpand      -E "$l" -e "$l" -p $priv        \
                        -c "$a file expansion"          \
                        $nattr  "$a" "$host" "" "$domain" )
    postzapDSNnotify a
    returns $a
fi
```

21.5.42. `login2uid`

Syntax:

login2uid *`username`*

Options:

none

Return Values:

"nobody-uid"

Either uid of "nobody", or no lookup match for given userid.

uid

UID of user.

Notes:

Prints the uid associated with the specified account name, if any. A side-effect is to add the GECOS name field of the account to the *fullname* in-core database, to add the login name to uid mapping to the *pwnam* in-core database, and to add the uid to login name mapping to the *pwuid* in-core database.

21.5.43. `lreplace`

Syntax1:

lreplace `varname indexnum anyvalue`

Syntax2:

lreplace `varname fieldname anyvalue`

Return Values:

none

Options:

none

Notes:

This replaces designated field on *varname* variable containing list-like data with *anyvalue* value.

The field designation can be given in numeric form, where the field index can be numeric (first field is zero), or *keyname* on key/value pair list.

If a key is not found (with key/value pairs), then designated pair is added to the list.

```
z$ tt=(aa 11 bb 22 cc 33 dd 44)
z$ grind $tt
(aa 11 bb 22 cc 33 dd 44)
z$ lreplace tt bb zz
```

```
z$ grind $tt
(aa 11 bb zz cc 33 dd 44)
z$ lreplace tt 0 aaa
z$ grind $tt
(aaa 11 bb 22 cc 33 dd 44)
z$ lreplace tt zz aa
z$ grind $tt
(aaa 11 bb 22 cc 33 dd 44 zz aa)
```

This is an example of "`lreplace`" use in the scripts:

```
# Usage: newattribute <oldattribute> <key1> <value1> [ <key2> <value2> ] ...
#
# Returns a new attribute list symbol with the <keyN> <valueN>
# attributes added to the contents of the <oldattribute> list.

newattribute (oldattribute) {
        local a null value

        a=$(gensym)
        eval $a=\$$oldattribute
        while [ "$#" != 0 ];
        do
                lreplace $a "$1" "$2"
                shift ; shift
        done
        echo -- "$a"
}
```

A fragment of code from inside the `crossbar.cf` shows different usages:

```
....
    usenet)
        lreplace from 2 "$(uucproute "$(user $from)")"
        tsift $(user $from) in
        $hostname!.*    ;;
        .*
            lreplace from 2 $hostname!$(user $from)
            ;;
        tfist
        # newsgroup name only
        lreplace to 2 "$(localpart "$(user $to)")"
        ;;
....
```

21.5.44. `malcontents`

Syntax:

```
malcontents
```

Return Values:

none

Options:

none

Notes:

ZMSH Debugging thing.

21.5.45. `printaliases`

Syntax:

printaliases [`-v`] [`-o` *`indexoutputfile`*] *`file`*

Return Values:

0

1

Error.

Options:

`-v`

Verbose.

`-o` *`indexoutputfile`*

Each header line will also generate a "header TAB byteoffset" line in the indexfile.

Notes:

This function was used by the "*newaliases(1)*" program to generate the aliases database from a source file.

That task has been moved into "***zmailer newdb***" (at Section 24.1.8) process, along with all other router database refreshment tasks.

Reads RFC822 syntax header lines from the specified file, parses them assuming contents must be an address list, and sorts and prints the header lines with all addresses in normal form. Comments are allowed; they extend from the character "#" at the beginning of a line, or after an address, to the end of line.

21.5.46. `process`

Syntax:

process *`messagefile`*

Return Values:

0

Successfull processing.

100

File name parameter missing.

other

Return value from the underlying scripts.

Options:

none

Notes:

The protocol switch function. It is called by the "*`daemon`*" function (at Section 21.5.14) to process a message found in the `$POSTOFFICE/router/` directory.

This function will in turn call an internal protocol-specific function which knows the syntax and semantics of the message file. The current version knows about messages submitted using the `MSG_RFC822` parameter to *mail_open(3)*. For that case, the protocol function is called "`rfc822`".

router has a bit more complex directory semantics, than is stated above. See "*zmailer(3)*" for details.

Although the "`process`" function is provided built in, it is usually overridden by a defined function in the *router* configuration file. (See `process.cf`, and the entrypoint text at Section 21.4.1)

21.5.47. `read`

Syntax:

read *`variable`*...

Return Values:

0

Successful read.

1

Error. (End of input)

64

Error. Missing mandatory variable name. (Usage.)

Options:

none

Notes:

The `read` will get one line of input, and if there are more than one variable name parameter, split it at the whitespaces (IFS chars).

There appears to be a bug when there are less IFS separated sequences than there are varnames: The extra varnames do get things at a bit random.. (2001-Oct-15)

If input line is zero length (or all IFS chars), the `read` will read another input line.

If the input line ends with character: "\" as its very last, the line gets a catenation of next input line effectively unfolding multiline coded string.

```
z# echo "1 2 3 4 5" | read v1 v2
z# echo $?
0
z# echo $v1
1
z# echo $v2
2 3 4 5
z# echo "11111" | read v1 v2
z# echo $?
0
z# echo $v1
11111
z# echo $v2
11111
```

21.5.48. `recase`

Syntax:

recase [-u | -l | -p] -- *string*

Return Values:

0

Success, `stdout` gets the result string.

64

Bad option. (Usage.)

Options:

--

End options.

-u

Convert into uppercase.

-l

Convert into lowercase.

-p

Prettify.

Notes:

A case-mapping function that prints the parameter string in either all-uppercase, all-lowercase, or capitalized (pretty).

The input to be converted is expected to be single string, e.g. not multiple strings.

Due to internal use of "getopt" for parameter pickup, if there is even the slightest change that the string begins with a hyphen (minus) character, invocation must use the -- pair to end the options, and to protect the parameter.

Typical use in the router scripts:

```
lcuser="$(recase -l -- "$user")"
```

21.5.49. `recipient`

Syntax:

recipient

Return Values:

1

Is a recipient address.

0

Is not.

Options:

none

Notes:

A boolean function that returns the value of the statement "executing a header rewriting function and the address is a recipient address in a message header".

21.5.50. `relation`

Syntax:

Generic:

relation `[-i]` `[-T]` `-t` *`dbtype`* `[/`*`subtype`*`]` `[-f` *`file`*`]` `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` `[-C` *`configfile`*`]` *`name`*

The separator character between "dbtype" and "subtype" can be either a comma (",") or a slash ("/"), as user wishes. In case the subtype is a filepath (or otherwise begins with a slash), the user propably wants to use comma to reduce confusion.

More specific versions:

relation `-t yp,`*`yp-mapname`* `-f` *`yp-domain`* `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` *`name`*
relation `-t bind/`*`query-type`* [-f *`file`*] [-e *#*] [-s *#*] [-:%blmnu] [-d *`driver`*] *`name`*
relation `-i -t ordered,`*`filepath2`* `-f` *`filepath1`* `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` *`name`*
relation `-i -t unordered,`*`filepath2`* `-f` *`filepath1`* `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` *`name`*
relation `-t ordered -f` *`filepath`* `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` *`name`*
relation `-t unordered -f` *`filepath`* `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` *`name`*
relation `-t` *`dbtype`* `[-f` *`file`*`]` `[-e` *`#`*`]` `[-s` *`#`*`]` `[-:%blmnu]` `[-d` *`driver`*`]` *`name`*

Note: "**zmailer newdb**" does not support "`-i`" option use! (FIXME! FIXME! 2001-Mar-8)

Special support for testing of presence of support for given *`dbtype`*:

relation `-T -t` *`dbtype dummy_name`*

Options are listed below. `name` is the name of the relation that is wanted to be created. Examples:

```
relation -t bind,mx mxhost
relation -t ordered -f $MAILVAR/db/routes routes
relation -t ordered -b -f /usr/lib/news/active.sorted newsgroups
```

Return Values:

0

Relation is reated successfully, however possible database access are not tried yet.

1..7

Error.

Options:

`-T`

Special flag enabling script to test for given db-type:

```
relation -T -t btree dummy && echo "have BTREE database"
```

`-t` *`dbtype`*`[,`*`subtype`*`]`

Below is a table to option interdependencies as they apply with different database types.

Figure 21-1. `relation`'s option interdependencies

Db-Type	Subtype meaning	`-f` option value
incore	ignored	ignored
header	ignored	ignored
selfmatch	ignored	ignored
hostsfile	ignored	ignored (?)
ordered	ignored (without `-i` option)	path-to-file
unordered	ignored (without `-i` option)	path-to-file
ordered	path-to-file-2 (with `-i` option)	path-to-file-1
unordered	path-to-file-2 (with `-i` option)	path-to-file-1
bind	DNS-query-subtype	ignored
yp	YP-mapname	YP-domain
ldap	ignored??	path-to-cfg-file
dbm	ignored	basepath-to-db-file
ndbm	ignored	basepath-to-db-file
gdbm	ignored	basepath-to-db-file
btree	ignored	basepath-to-db-file
bhash	ignored	basepath-to-db-file

One of the known types of databases, currently:

`incore`

A database maintained in virtual memory (using splay trees). This type should not be used for any database that must periodically be flushed, since all occupied memory can be freed.

`header`

A special incore database type used to store RFC822 header semantics information. It is unlikely to be used for anything else.

`ordered`

The `-f`-option defines the path of the file.

A file with key-value pairs on every line, separated by whitespace, sorted by key. (See **sort**(1).)

```
key_at_line_start   data at the same line
```

`unordered`

The `-f`-option defines the path of the file.

A file with key-value pairs on every line, separated by whitespace.

```
key_at_line_start   data at the same line
```

`ordered,`*`path-to-file-2`*

The `-f`-option defines the path of the file-1.

The version for antique "`-i`" mode.

`unordered,`*`path-to-file-2`*

The `-f`-option defines the path of the file-1.

The version for antique "`-i`" mode.

`hostsfile`

The `-f`-option defines the path of the file. (In theory...)

A rather theorethical database looking into *hosts(5)* database file (often `/etc/hosts`).

`bind/query-subtype`

The `-f`-option defines the path of the resolver configuration file. (Not implemented!)

The BIND implementation of a Domain Name System resolver. The subtype for this type is the name of a Resource Record type in the `IN` class.

Supported subtypes are: A, AAAA, ANY, CNAME, MX, MXLOCAL, MXWKS, PTR, TXT, UINFO, WKS

FIXME! FILL DETAILS! (About the return values)

`ndbm`

The `-f`-option defines the base path of the files, and the NDBM appends ".pag" and ".dir" to each file.

The newer DBM as created at (I think) BSD 4.2. This is two-file database with API utilizing nonglobal API, that is, multiple databases can be open simultaneously. This appends .dir and .pag to the supplied name!

Limitation: The length of *key* plus the length of *data* must not exceed 1024 bytes. With certain kinds of alias databases this may be too low limit!

`dbm`

The `-f`-option defines the base path of the files, and the DBM appends ".pag" and ".dir" to each file.

The old ATT DBM library with even worse limitations than `ndbm` has. Avoid if you can. (ZMailer can manage with this also, each lookup is done by opening the DB, and closing immediately afterwards.)

Some versions of ATT DBM *did not* contain externally callable `close`() function! ZMailer propably won't work at such a system anyway...

`gdbm`

The `-f`-option defines the path of the database file, the GDBM does not append anything to the name.

The GNU implementation of the new DBM library. Note: GDBM uses one file, which is named exactly as you parametrize it. This is unlike NDBM, which appends .dir and .pag to the supplied name!

`yp,`*`mapname`*

The Network Information Service from Sun Microsystems Inc. (Later renamed to be NIS, the still newer NIS+ is not supported).

The *`mapname`* "subtype" passes knowledge about which YP-map the query is to be done from.

The `-f`-option is used to pass the *YP-domain* information to the interface.

`btree`

SleepyCat DB 1.x, 2.x, 3.x or 4.x B-Tree database

The `-f`-option defines the path of the database file, the SleepyCat DB does not append anything to the name. (This is true with versions 1.x, 2.x, 3.x, and 4.x.)

The `-C`-option defines SleepyCat DB environment configuration file, which can be used to define advanced features, mainly *Concurrent Data Store* function.

```
 FIXME! FIXME! config file!

$ cat /opt/mail/db/sleepyenv.conf
#
# SleepyCat DB 3/4 environment settings
#

envhome  = /opt/mail/db
#tmpdir = ...
envmode  = 0600
envflags = CDB, CREATE, RO
```

`bhash`

SleepyCat DB 1.x, 2.x, 3.x or 4.x HASH database

The `-f`-option defines the path of the database file, the SleepyCat DB does not append anything to the name. (This is true with versions 1.x, 2.x, 3.x, and 4.x.)

The `-C`-option defines SleepyCat DB environment configuration file, which can be used to define advanced features, mainly *Concurrent Data Store* function.

```
 FIXME! FIXME! config file!

$ cat /opt/mail/db/sleepyenv.conf
#
# SleepyCat DB 3/4 environment settings
#

envhome  = /opt/mail/db
#tmpdir = ...
envmode  = 0600
envflags = CDB, CREATE, RO
```

`selfmatch`

Given address literal *without* wrapping square brackets, this "database" decodes the address, and checks if presented IP address is one used by the system at the moment.

`ldap`

FIXME! WRITEME!

`-f` *`file`*

A file associated with the database, typically the file containing the data, or the basename of DBM files or something similarly relevant to the database access routine.

`-e` *`#`*

The default time-to-live on cached information. When the information has been in the cache for this many seconds, it is discarded. The default is 0.

`-s` *`#`*

Sets the cache size to the specified number. The default is usually 10, depending on the database type.

`-b`

If the key exists in the database, return the key as the value. ("Boolean relation")

`-i`

If the key exists, its value is a byte offset into a file named by the subtype for this database. The value then becomes the concatenation of the data on the lines following that offset which start with whitespace.

This was used for the aliases file back in early 1990, and is usable only with `ordered`, and `unordered` database types.

(FIXME! IMPLEMENT? To think of it, this makes eminently sense also for `dbm`, and `ndbm` which have data size limitations. But then, SleepyCat DB is recommended for internal databases anyway.)

`-l`

Map all keys to ASCII lowercase before searching.

`-m`

Check for file content modification before every access. Reopen the file when a change is detected.

This option is used when the **router** should discover changes to a database underfoot so it need not be restarted to use new data.

This is recommended on relations which use unordered, or ordered datasets (aliases, routes, ...), and especially if the system is configured to use `mmap(2)` facility. Updating such databases should preferably use **mv** command to move a new version of the database in place of the old one.

`-n`

If the key exists in the database and the value is null or list, return the key as value. Otherwise return the value retrieved, if any.

`-u`

Map all keys to ASCII uppercase before searching.

`-d` *[pathalias|pathalias.nodot|longestmatch]*

Specifies a search driver that allows searching for structured keys using special knowledge. The argument to this option must be a known driver.

FIXME! WRITEME! WRITE MORE!

`-%`

We shall do positional parameter substitutions ("%0" thru "%9") on database lookup result data. [XREF??]

The **zmailer newdb** configuration file `$MAILVAR/dbases.conf` uses presence of "%" to signal this aspect of relation wrapper generation. [XREF??]

`-:`

Actually this is *ignored* if present, the **zmailer newdb** configuration file `$MAILVAR/dbases.conf` uses presence of ":" to signal certain aspects of relation wrapper generation. [XREF??]

Notes:

On systems with `USE_MMAP` the ordered, and unordered databases are r/o mapped into memory, and for ordered case, a special line-index is generated for speeding up the binary search. (Makes less system calls that way.)

21.5.51. `rest`

Syntax:

rest *`object-reference`*

Return Values:

Pointer to copied lisp-object of object-reference's *`cdr`* operation.

Options:

none

Notes:

This command is synonymous to the *`cdr`* command (at Section 21.5.11).

21.5.52. `return`

Syntax:

return ***`lisp-object`***

Return Values:

lisp-object

The argument lisp-object contains a non ASCII digit character, or is a complex lisp-object.

string

The argument lisp-object contains a non ASCII digit character, and is a simple string-object.

statuscode

The argument lisp-object contains a all ASCII digit characters, and is a simple string-object.

See the Notes below about this, too.

NULL

Invalid lisp-object.

Options:

none

Notes:

The system has a weird dichtomy on returning numeric vs. other results.

Presume a function call with two different possible results, failure indication, and successfull (arbitrary) string result:

```
tmp=$(funcnnn args..) && returns $tmp
return $tmp  # error code return!
```

21.5.53. `returns`

Syntax:

returns ***`lisp-object`***

Return Values:

lisp-object

This one will always return the lisp object without interpreting possible string value to be numeric return code.

Options:

none

Notes:

See notes of "`return`" above.

21.5.54. `rfc822`

Syntax:

rfc822 *`messagefile`*

Return Values:

status

Options:

none

Notes:

This function controls the parsing and processing of the message file in RFC822/976/2822 format. It is called by the *`process`* function (at Section 21.5.46). .

21.5.55. `rfc822date`

Syntax:

rfc822date

Return Values:

0

Side effect: `stdout` gets current time string printed in RFC822/2822 format.

Options:

none

Notes:

Prints the current time in RFC822/2822 format.

21.5.56. `rfc822syntax`

Syntax:

rfc822syntax *`address`*

Return Values:

0

Given input matches RFC 822/976/2822 for "route-address" syntax specification.

1

Error. Given input is syntactically somehow invalid.

64

Argument count is not exactly 1. (Usage.)

Options:

none

Notes:

This is a simple interface to the address parser. If the command line argument is a syntactically valid RFC822/976/2822 address, this command is silent and returns 0 as status. If there is a parse error, a verbose error message is printed to `stdout` and the function yields a non-zero return status.

21.5.57. `runas`

Syntax:

runas *`user function`* [*`arguments`*...]

Return Values:

Any of the values yielded by the executed "function", or:

0

Internal evaluation of "function" did yield 0.

1

Setting target uid failed.

64

Mandatory parameters missing. (Usage.)

abort

Resetting target uid to system uid (root) failed.

Options:

none

Notes:

Changes the current effective user id of the *router* process to that given (which may be numeric or an account name), then runs the specified function with the specified arguments, then switches the effective user id of the process back (to root).

21.5.58. `sender`

Syntax:

sender

Return Values:

1

Is a sender address.

0

Is not sender address.

Options:

none

Notes:

A boolean function that returns the value of the statement "executing a header rewriting function and the address is a sender address in a message header".

21.5.59. `set`

Syntax:

set [`-a` | `-e` | `-f` | `-h` | `-n` | `-t` | `-u` | `-v` | `-x` | `-L` | `-C` | `-P` | `-S` | `-k`] [`-`] [*`variable`*]

Without parameters `set` prints variable values.

Return Values:

Pointer to copied structure of `car` operation.

Options:

`-a`

Automatically export changed variables.

`-e`

Exit on error exit status of any command.

`-f`

Disable filename generation (no globbing).

`-h`

Hash program locations.

`-n`

Read commands but do not execute them.

`-t`

Read and execute one command only.

`-u`

Unset variables are error on substitution.

`-v`

Print shell input lines as they are read.

`-x`

Print commands as they are executed.

`-L`

Trace LEXER processing (sslWalker).

`-C`

Print branch and emit inputs (sslWalker).

`-P`

Trace execution (sslWalker).

`-S`

Print input buffers when used (sslWalker).

`-k`

Not supported option.

`-`

Do nothing.

Notes:

none

21.5.60. `shift`

Syntax:

shift [number]

Return Values:

0

Success.

1

Error, out of parameters to shift.

Options:

none

Notes:

Modifies caller's argument vector by shifting left one (or specified number) of argument(s) in current `ARGV`.

21.5.61. `sleep`

Syntax:

sleep *`number`*

Return Values:

0

Did sleep a bit, does not tell is anybody interrupted the sleep.

64

Missing mandatory integer argument. (Usage.)

Options:

none

Notes:

Does not tell if the sleep has been interrupted somehow.

21.5.62. `squirrel`

Syntax:

squirrel [-]event

Return Values:

0

1

Error.

Options:

`-`

Set flag value to 0.

(none)

Set flag value to 1.

The events are:

- `breakin`
- `badheader`
- `illheader`
- `nochannel`
- `nosender`

Notes:

Sets the kinds of events that cause a message to be copied into the `$POSTOFFICE/postman/` directory. Whether or not a "`-`" is necessary for an event depends on the current state of the event's flag.

The usage message will indicate what to do to toggle the event flag:

```
z# squirrel
Usage: squirrel [ breakin | -badheader | illheader | nochannel | nosender ]
```

21.5.63. `stability`

Syntax:

stability [`on` | `off`]

Return Values:

0

Did the work successfully.

64

Bad parameters. (Usage.)

Options:

none

Notes:

Determines whether the *router* will process incoming messages in arrival order (when on), or in random order determined by position in the router directory. The *router* will by default do the first queue scan in stable mode, and subsequent scans in unstable mode. The name of this command is the name for a similar characteristic of sorting algorithms.

21.5.64. "`test`", "`[`"

See: "*`[`*", a.k.a. "*`test`*" at Section 21.5.3.

21.5.65. `times`

Syntax:

times

Return Values:

0

Prints to `stdout` the spent usermode time for process itself, and to all of its children.

1

Error in `times`(2) system call.

64

Usage error (no parameters allowed.)

Options:

none

Notes:

Prints to `stdout` the spent usermode time for process itself, and to all of its children.

Sample output:

```
12m33s  22m59s
```

21.5.66. `trace`

Syntax:

trace key1 ... keyN
untrace key1 ... keyN

Enables tracing of the specified items. The valid keywords are listed in the options below.

Return Values:

0

Successful setting/clearing.

64

Parameter name error. (Usage.)

Options:

`all`

Turns on all tracing options.

You only do this to test the I/O capabilities of your system. (`rfc822`, and `regexp` options generate *a lot of output!*)

`assign`

Prints shell variable assignments.

`bind`

Prints various information from the code that calls the DNS resolver.

`compare`

Prints `*sift` statement pattern-selector comparisons.

`db`

Prints database lookups, including cache search and update information.

`except`

Inverts the sense of what is being done; e.g.:

```
trace all except rfc822 regexp
```

which is (nearly) exquivalent of:

```
trace all
untrace rfc822 regexp
```

`final`

Prints the message envelope information after processing each message.

`functions`

Prints shell function calls and return values, with nesting indicated by indentation.

`matched`

Prints `*sift` statement pattern-selector matches.

`memory`

Prints memory allocation information after each message.

`on`

Same as `functions` -option.

`regexp`

Prints regular expression matching execution.

`resolv`

Turns on the `RES_DEBUG` flag in the *BIND* resolver library, and prints various information from the code that calls the DNS resolver.

`rewrite`

Prints the tokenized addresses sent through the message header address rewriting functions.

`router`

Prints the tokenized addresses sent through the `router` function.

`sequencer`

Prints the procedural steps taken during message processing.

Notes:

Authors most common "`trace`" incantation has been wrapped into standard script routine:

```
#|
#|  I kept typing in this trace command so frequently, that eventually
#|  I just had to make for it into a single command... /Matti Aarnio
#|

rtrace () {
        trace all except rfc822 regexp
}
```

21.5.67. `trap`

Syntax:

trap [[*script trap_nro*] ...]

Return Values:

0

Set successfully, or displayed successfully.

Options:

none

Notes:

If no parameters are given, `trap` prints all known traps.

In all aspects this is quite alike any bourne-shell "`trap`" function.

Set, and unset a trap:

```
trap "db flush aliasesdb ; log flushed aliases" 16
trap "" 16
```

21.5.68. `type`

Syntax:

type [*command*...]

Return Values:

0

Always returns this value.

The `stdout` gets the report.

Options:

none

Notes:

```
z# type trap
```

```
trap is a shell builtin
z# type foobar
foobar not found
z# foobar () {echo foo}
z# type foobar
foobar is a shell function
z# type rfc822
rfc822 is a shell builtin
z# type process
process is a shell function
z# echo $?
0
z# type no-such-thing
no-such-thing not found
z# echo $?
0
z# type
z# echo $?
0
```

21.5.69. `uid2login`

Syntax:

uid2login uid

Return Values:

0

Argument count ok, and "*`uid`*" begins with a digit.

The `stdout` gets the username.

64

Parameter error. (Usage.)

Options:

none

Notes:

Prints the first account name associated with a specified numeric user id, if any, or "*`uid#uid`*" if no account exists with that user id. It has the same side-effects as the *`login2uid`* function (at Section 21.5.42).

21.5.70. `umask`

Syntax:

umask [octal-number-mask]

Return Values:

0
: Successfull printing of the octal value, or setting new umask(2) value.

64
: Parameter error. (Usage.)

Options:

none

Notes:

Without parameters the new default mask is 077, and old is printed.

21.5.71. `unset`

Syntax:

unset [variable...]

Return Values:

0
: Had enough parameters, executed something, and possibly complained something to "`stderr`".

64
: Missing mandatory (at least one) argument. (Usage.)

Options:

none

Notes:

This throws away named variables from all variable scopes.

```
z# echo $TERM
xterm
z# unset TERM
z# echo "'$TERM'"
''
```

21.5.72. `untrace`

Syntax:

trace key1 ... keyN
untrace key1 ... keyN

Disables tracing of the specified items. This is inverse of *`trace`* (at Section 21.5.66).

Return Values:

0

Successfull clearing/setting

64

Parameter name error. (Usage.)

Options:

See the *`trace`* function (at Section 21.5.66) for valid keywords.

Notes:

See the *`trace`* function (at Section 21.5.66) for valid keywords.

21.5.73. `user`

Syntax:

user *`object-reference`*

Return Values:

The next-address (3rd) component of and address quad.

Options:

none

Notes:

This is essentially same as:

```
$(cdr $(cdr $(cdr $addrquad)))
```

21.5.74. `wait`

Syntax:

wait [*pid*]

Return Values:

Besides of the return codes of processes being waited, this can yield:

0

no more processes

64

Bad parameters. (Usage.)

Options:

none

Notes:

none

Chapter 22. Scheduler Reference

```
... deeper details of internal protocols and algorithms
 - Configuration Language Syntax Details (?)
 - Resource Management
 - What and how  scheduler.auth  can be tuned
   - Security issues
 - Diagnostics reporting, canned messages (forms/* files)
 - (MAILQv1/)MAILQv2 protocol for MAILQv2 client writer
 - Scheduler-TA interface
```

The **scheduler** daemon manages the delivery processing of messages in ZMailer.

The **router** creates message control files in the `$POSTOFFICE/transport/` directory. These refer to the original message files in the `$POSTOFFICE/queue/` directory.

The **scheduler** reads each message control file from `$POSTOFFICE/transport/`, translates the contained message and destination information into internal data structures.

Based on scheduling, priority, and execution information read from a configuration file, the **scheduler** arranges to execute *Transport Agents* relevant to the queued messages.

At the time scheduled for a particular transport agent invocation, the **scheduler** will start a transport agent (or use one from idle-pool), and tell it one by one which message control files to process. When all the destination addresses in a message have been processed, the **scheduler** performs error reporting tasks if any, and then deletes the message control file in `$POSTOFFICE/transport/` and the original message file in `$POSTOFFICE/queue/`.

All message delivery is actually performed by *Transport Agents*, which are declared in a configuration file for the **scheduler**. Each transport agent is executed with the same current directory as the **scheduler**. The *scheduler-transporter* interaction protocol is described later.

The standard output of each transport agent are destination address delivery reports; either successful delivery, unsuccessful delivery, or deferral of the address. Each report uses byte offsets in the message control file to refer to the address. Reports may also include a comment line which will be displayed in the reports of the **scheduler**.

Two types of reports are produced:

1. Error messages caused by unsuccessful delivery of a message are appended to its message control file. Occasionally, for example, when all addresses have been processed, the **scheduler** generates an error message to the error return address of the message (usually the original sender).
2. The **scheduler** binds itself to a well-known TCP/IP port (*MAILQ, TCP port 174*) on startup. Any connections to this port are processed synchronously in the **scheduler** at points in the execution where the state is internally consistent. The **scheduler** simply dumps its internal state in a terse format to the TCP stream. It is expected that the client program will reconstruct the data structures sufficiently to give a user a good idea of what the scheduler thinks the world looks like. The *mailq*(1) program serves this purpose.

Usage:

Invoking **scheduler** without any parameter will start it as a daemon.

scheduler [-dinvFHQSVW] [-E newentsmax] [-f configfile] [-l statisticslog] [-L logfile] [-N transpmaxfno] [-p channel/host-pair] [-P postoffice] [-q rendezvous] [-R maxforkfreq]

Parameters:

`-d`

run as a daemon, usually used after -v to log daemon activity in great detail.

`-E` *`newentsmax`*

when globbing new tasks from the directory, pick only first *`newentsmax`* of them, and leave rest for a latter scan run.

`-f` *`configfile`*

overrides the default configuration file `$MAILSHARE/scheduler.cf`.

`-F`

Freeze -- don't actually run anything, just do queue scanning. (For debug purposes..)

`-H -HH`

Use multi-level hashing at the spool directories. This will efficiently reduce the lengths of the scans at the directories to find some arbitrary file in them. One "`H`" means *single level hashing*, two "`HH`" mean *dual level hashing*. *Hash* is directory which name is single upper case alphabet (A-Z).

`-i`

run interactively, i.e., not as a daemon.

`-l` *`statisticslog`*

starts the appending of delivery statistics information (ASCII form) into given file. No default value.

`-L` *`logfile`*

overrides the default log file location `$LOGDIR/scheduler`.

`-n`

Toggles the configuration flag called "default_full_content", which defines what will be "DSN RET" parameter assumed value in case the originator didn't supply that parameter.

The default behaviour is similar to "RET=FULL", while usage of this option is equivalent of "RET=HDRS".

This option does not override originator supplied DSN RET parameter value.

`-N` *`transmaxfno`*

sets how many filehandles are allocated for the **scheduler**'s started children (if the system has adjustable resources.)

`-p` *`channel/host`*

A debug option for running selectively some thread under a single instance of the scheduler.

Use this option with “`-v`”.

`-P` *`postoffice`*

specifies an alternate `$POSTOFFICE/` directory.

`-q` *`rendezvous`*

the rendezvous between machines without TCP/IP networking,

Scheduler and **mailq(1)** is done using a well-known named pipe. This option overrides the default location for this special file, either `$RENDEZVOUS` or `/usr/tmp/.mailq.text`. (not used in real life; aspect of ZMailer’s support for low-tech things..)

`-Q`

The “`Q`”-mode, don’t output the old style data into the queue querier, only the new-style one.

`-S`

Synchronous startup mode, scans all jobs at the directory before starting even the first transporter.

`-v`

Verbose logging in interactive mode -- for debug uses.

`-V`

Print version, and exit

`-W`

Another option for debugging, delay the start of the verbose logging until after all jobs have been scanned in, and it is time to start the transporters.

22.1. Configuration Language

`\index{{\tt scheduler.conf} file}\index{scheduler, {\tt scheduler.conf}}`

The **scheduler** configuration file consists of a set of clauses.

There are two kinds of clauses:

- PARAM-entries
- Group-Clause selections

22.1.1. PARAM-entries

There are three kinds of PARAM entries, all of them start at the column number 0 (left edge):

```
#
# MAILQv2 authentication database file reference:
# If you define this (like the default is), and the file exists,
# scheduler mailq interface goes to v2 mode.
# (Nonexistence of this file  A) leaves system running, B) uses MAILQv1
#  interface along with its security problems.)
#

PARAMauthfile = "${MAILSHARE}/scheduler.auth"

#PARAMmailqsock = "UNIX:/path/to/mailq.sock"
#PARAMmailqsock = "TCP:174"

# Time for accumulating diagnostic reports for a given message, before
# all said diagnostics are reported -- so that reports would carry more
# than one diagnostic in case of multi-recipient messages.
#PARAMglobal-report-interval = 15m
```

The *PARAMauthfile* defines Scheduler's MAILQv2 authentication file; more at Section 22.3.

The *PARAMmailqsock* defines non-standard socket for the MAILQv2, the default is "TCP:174" meaning local host binding on wild-card address, and port 174 of TCP. Other ports and protocols can be set. *The **mailq** tool will not parse this file to know where to connect.*

The *PARAMglobal-report-interval* is how often (or infrequently) to run the scheduler's subtask of reporting so far accumulated diagnostics. Original behaviour was to report diagnostics only when message timed out, or last recipient was otherwise disposed of. Current method is *somewhat* quicker.

22.1.2. Group-Clause selection

Each clause is selected by the pattern it starts with. The patterns for the clauses are matched, in sequence, with the *channel/host* string for each recipient address. When a clause pattern matches an address, the parameters set in the clause will be applied to the *scheduler*'s processing of that address. If the clause specifies a command, the clause pattern matching sequence is terminated.

This is a clause:

```
local/* interval=10s
        expiry=3h
        # want 20 channel slots in case of blockage on one
        maxchannel=20
        # want 20 thread-ring slots
        maxring=20
        command="mailbox -8"
```

A clause consists of:

- A selection pattern (in shell style) that is matched against the *channel/host* string for an address.
- 0 or more variable assignments or keywords (described below).

If the selection pattern does not contain a “/”, it is assumed to be a channel pattern and the host pattern is assumed to be the wildcard “*”.

22.1.3. Clause components

The components of a clause are separated by whitespace. The pattern introducing a clause must start in the first column of a line, and the variable assignments or keywords inside a clause must not start in the first column of a line. This means a clause may be written both compactly all on one line, or spread out with an assignment or keyword per line.

If the clause is empty (i.e., consists only of a pattern), then the contents of the next non-empty clause will be used.

The typical configuration file will contain the following clauses:

- a clause matching all addresses (using the pattern “*/*”) that sets up default values.
- a clause matching the local delivery channel (usually “local”).
- a clause matching the deferred delivery channel (usually “hold”).
- a clause matching the error reporting channel (usually “error”).
- clauses specific to the other channels known by the **router**, for example, “smtp” and “uucp”.

The actual names of these channels are completely controlled by the **router** configuration file.

Empty lines, and lines whose first non-whitespace character is “#”, are ignored.

Variable values may be unquoted words or values or doublequoted strings. Intervals (delta time) are specified using a concatenation of: numbers suffixed with ‘s’, ‘m’, ‘h’, or ‘d’ modifiers designating the number as a second, minute, hour, or day value. For example:

```
1h5m20s
```

22.1.4. Variables and keywords

The known variables and keywords, and their typical values and semantics are:

`ageorder`

Default is to randomize the order of tasks at the queue, when it is started, with this the order is that of the original spool-file MTIME. Oldest first.

`bychannel`

is a keyword (with no associated value) that tells the **scheduler** that the transport agent specified in the command will only process destination addresses that match the first destination channel it encounters. This is automatically set when the string “`$channel`” occurs in the command, but may also be specified manually by this keyword. This is rarely used.

`command="sss"`

is the command line used to start a transport agent to process the address. The program pathname is specified relative to the `$MAILBIN/ta/` directory.

The string “`$channel`” is replaced by the current matched channel, and “`$host`” replaced by the current matched host, from the destination address.

It is strongly recommended that the “`$host`” is not to be used on a command definition, as it limits the re-usability of idled transporter.

It is possible to place environment-string setting statements into the beginning of the line:

```
command="MALLOC_DEBUG_=1 OTHER=var cmdname cmdparams"
```

`expiry=nn` (3d)

specifies the maximum age of an address in the **scheduler** queue before a repeatedly deferred address is bounced with an expiration error. The actual report is produced when all addresses have been processed.

`group="sss"` (daemon)

is the group id of a transport agent processing the address. The value is either numeric (a gid) or a group name.

`idlemax=nn` (3x interval)

When a transport agent runs out of jobs, they are moved to *idle pool*, and if a TA spends more than `idlemax` time in there, it is terminated.

`interval=nn` (1m)

specifies the primary retry interval, which determines how frequently a transport agent should be scheduled for an address. The value is a delta time specification. This value, and the `retries=...` value mentioned below, are combined to determine the interval between each retry attempt.

`maxchannel=nn` (0)
(`maxchannels=nn`)

if retrying an address would cause the number of simultaneously active transport agents processing mail for the same channel to exceed the specified value, the retry is postponed. The check is repeated frequently so the address may be retried as soon as possible after the scheduled retry interval. If the value is 0, a value of 1000 is used instead.

`maxring=nn` (0)
(`maxrings=nn`)

Recipients are groupped into *threads*, and similar threads are groupped into *thread-rings*, where same transport agent can be switched over from one recipient to another. This defines how many transport agents can be running at any time at the ring.

`maxta=nn` (0)

if retrying an address would cause the number of simultaneously active transport agents to exceed the specified value, the retry is postponed. The check is repeated frequently so the address may be retried as soon as possible after the scheduled retry interval. If the value is 0, a value of 1000[1] is used instead.

`maxthr=nn` (1)

> This limits the number of parallel transport agents within each thread; that is, using higher value than default “1”

`nice=nn`

> Defines *relative* priority value for transport-agent process. Default is not to use this. Value range in between -40 to 40.

`overfeed=nnn` (0)

> Max number of tasks to feed from the thread to the transporter agent when feeding jobs to it. The **scheduler** main-loop at the `mux()` is a bit sluggish, thus with this we can keep the transporters busy.
>
> The default is defined at the */* clause.

`priority=nn`

> Defines *absolute* priority value for transport-agent process. Default is not to use this. Value range in between -20 to 20.

`queueonly`

> a clause with *queueonly* flag does not auto-start at the arrival of a message, instead it must be started by means of **smtpserver(8)** command **`ETRN`** through an SMTP connection.
>
> To have message expiration working, following additional entries are suggested:
>
> ```
> interval=1h
> retries="24"
> ```
> That is, retry once in a day.

`reporttimes="n n n"` ()

> Placeholder for DELAYED reporting mechanism.

`retries="n n n"` (1 1 2 3 4 8 13 21 34)

> specifies the retry interval policy of the **scheduler** for an address. The value must be a sequence of positive integers, these being multiples of the primary interval before a retry is scheduled. The **scheduler** starts by going through the sequence as an address is repeatedly deferred. When the end of the sequence is reached, the **scheduler** will jump into the sequence at a random spot and continue towards the end. This allows various retry strategies to be specified easily:
>
> - brute force (or "jackhammer"):
>
>   ```
>   retries=0
>   ```
>
> - constant primary interval:
>
>   ```
>   retries=1
>   ```
>
> - instant backoff:
>
>   ```
>   retries="1 50 50 50 50 50 50 50 50 50 50 50 50"
>   ```
>
> - slow increasing (fibonacci) sequence:
>
>   ```
>   retries="1 1 2 3 5 8 13 21 34"
>   ```
>
> - s-curve sequence:
>
>   ```
>   retries="1 1 2 3 5 10 20 25 28 29 30"
>   ```
>
> - exponential sequence:

```
retries="1 2 4 8 16 32 64 128 256"
```

- etc.

`skew=nn` (5)

Leftover of earlier **scheduler** internal algorithms, does not make sense anymore.

`sysnice=nn`

Can be used (if desired) at the */* clause to set *relative niceness* for the *scheduler* process, and all of its children.

`syspriority=nn`

Can be used (if desired) at the */* clause to set *absolute priority* for the *scheduler* process, and all of its children.

`user="sss"` (root)

is the user id of a transport agent processing the address. The value is either numeric (a uid) or an account name.

`wakeuprestartonly`

Start only one instance of handling processes, never mind what other settings say.

For example, this is a complete configuration file:

```
# Default values
*/*     interval=1m expiry=3d retries="1 1 2 3 5 8 13 21 34"
        maxring=0 maxta=0 skew=5 user=root group=daemon
# Boilerplate parameters for local delivery and service channels
local/* interval=10s expiry=3h maxchannel=2 command=mailbox
error   interval=5m maxchannel=10 command=errormail
hold/*  interval=5m maxchannel=1 command=hold
# Miscellaneous channels supported by router configuration
smtp/*.toronto.edu
smtp/*.utoronto.ca maxchannel=10 maxring=2
        command="smtp -srl /var/log/smtp"
smtp/*  maxchannel=10 maxring=5
        command="smtp -esrl /var/log/smtp"
uucp/*  maxchannel=5 command="sm -c $channel uucp"
```

The first clause ("*/*") sets up default values for all addresses. There is no command specification, so clause matching will continue after address have picked up the parameters set here.

The third clause ("error") has an implicit host wildcard of '*', so it would match the same as specifying "error/*" would have.

The fifth clause ("smtp/*.toronto.edu") has no further components so it selects the components of the following non-empty clause (the sixth).

Both the fifth and sixth clauses are specific to address destinations within the TORONTO.EDU and UTORONTO.CA organization (the two are parallel domains). At most 10 deliveries to the smtp channel may be concurrently active, and at most 2 for all possible hosts within TORONTO.EDU. If "$host" is mentioned in the command specification, the transport agent will only be told about the message control files that indicate SMTP delivery to a particular host. The actual host is picked at random from the current choices, to avoid systematic errors leading to a deadlock of any queue.

22.2. Resource Management

For resource management there are following configuration attributes:

`maxta=nn`

Max number of transporter processes under the **scheduler**.

`maxchannel=nn`

Max number of processes with this same "channel".

`maxring=nn`

Max number of processes with this set of threads.

`maxthr=nn`

Max number of processes at any given thread in this set of threads.

`idlemax=time`

How long the non-active (idle) transporter processes are allowed to keep around.

`overfeed=nnn`

Max number of tasks to feed from the thread to the transporter agent when feeding jobs to it. The **scheduler** main-loop at the `mux()` is a bit sluggish, thus with this we can keep the transporters busy.

22.3. `scheduler.auth` file

The file whose default boilerplate is shown at Figure 22-1 is able to control what kind of things (and who, of those who know shared secrets) can ask the scheduler to do via the so called "MAILQv2" protocol.

Figure 22-1. Sample of "`scheduler.auth`" file

```
#
# APOP-like authentication control file for the ZMailer scheduler.
#
# Fields are double-colon (':') separated, and are:
#   - Username
#   - PLAINTEXT PASSWORD (which must not have double-colon in it!)
#   - Enabled attributes (tokens, space separated)
#   - Addresses in brackets plus netmask widths:  [1.2.3.4]/32
#
# Same userid CAN appear multiple times, parsing will pick the first
# instance of it which has matching IP address set
```

```
#
# The default-account for 'mailq' is 'nobody' with password 'nobody'.
# Third field is at the moment a WORK IN PROGRESS!
#
# SECURITY NOTE:
#   OWNER:      root
#   PROTECTION: 0600
#
# Attribute tokens:
#       ALL     well, a wild-card enabling everything
#       SNMP    "SHOW SNMP"
#       QQ      "SHOW QUEUE SHORT"
#       TT      "SHOW QUEUE THREADS", "SHOW THREAD channel/host"
#       ETRN    "START THREAD channel host"
#       KILL    "KILL THREAD channel host", "KILL MSG spoolid"
#
# - "nobody" via loopback gets different treatment from
#   "nobody" from anywhere else.
#
nobody:nobody:SNMP QQ TT ETRN:  [127.0.0.0]/8 [ipv6.0::1]/128
nobody:nobody:SNMP ETRN:        [0.0.0.0]/0   [ipv6.0::0]/0
#watcher:zzzzz:SNMP QQ TT ETRN: [127.0.0.0]/8 [192.168.0.1]/32
#root:zzzzzzz:ALL:              [127.0.0.0]/8 [192.168.0.2]/32
```

22.4. mailq protocol v.1

FIXME! FIXME!

Upon accepting a TCP connection on the MAILQ port (TCP port 174), the scheduler dumps data to the TCP stream in the following format and immediately closes the connection.

The TCP stream syntax is:

```
version id\n
data in id-dependent format<close>
```

The first line (all bytes up to an ASCII LF character, octal 12) is used to identify the syntax of all bytes fol- lowing the line terminator LF. The first 8 characters of the first line are “version” as a check that this is indeed a MAILQ port server that has been reached, the remaining bytes are the real data format identification. The data is interpreted according to that format until the terminating connection close.

Format identifiers should be registered with the author. The only one currently defined is “zmailer 1.0”. For that data format, the syntax of the data following the first LF is:

```
Vertices:\n
(<key>:\t><msg-file>\t><n-addrs>; <off1>(,<offN>)*\t>[#<text>]\n)*
(Channels:\n
(<word>:\t>(><key>)+\n)+
Hosts:\n
(<word>:\t>(><key>)+\n)+)?
```

Where:

`\n`

is an ASCII linefeed

`\t`

is an ASCII tab

`key`

is an unsigned decimal number

`msg-file`

is a contiguous string (it is the message file name relative to a known directory)

`n-addrs`

is an unsigned decimal number (number of addresses)

`off1...offN`

are unsigned decimal numbers (address byte offsets)

`text`

is a string not containing an ASCII linefeed (status message)

`word`

is a contiguous string (a “contiguous string” is a sequence of printable non-space characters

For example, here is sample output from connecting to the MAILQ port:

```
version zmailer 1.0
Vertices:
311424:37141; 116
311680:64722; 151,331#128.100.8.4: Null read! (will retry)
312192:63471; 152#128.89.0.93: connect: Connection timed out (will retry)
Channels:
smtp:>311424>311680>312192
Hosts:
scg.toronto.edu:>311424
mv04.ecf.toronto.edu:>311680
relay1.cs.net:>312192
```

This is sufficient information to be able to reconstruct the transport queues as seen by the scheduler process, and to find more information than what is shown here by actually looking up the message control and data files referred to.

22.5. mailq protocol v.2

FIXME! FIXME!

22.6. Transport Agent Interface Protocol

The transport agent interface follows a master-slave model, where the TA informs the scheduler that it is ready for the work, and then the scheduler sends it one job description, and waits for diagnostics. Once the job is finished, the TA notifies the scheduler that it is ready for a new job.

A short sample session looks like this:

```
(start the transport agent)
#hungry                 --> (TA to scheduler)
spoolid \t hostspec     <-- (scheduler to TA)
diagnostics             --> (TA to scheduler)
#hungry                 --> (TA to scheduler)
...
```

Normal diagnostic output is of the form:

```
  id / offset \t notarydata \t status message
```

where:

id

is the inode number of the message file,

offset

is a byte offset within its control file where the address being reported on is kept,

notarydata

is a *Ctrl-A* separated tuple is delivery-status-notification information for the message,

status

is one of:*ok, ok2, ok3, error, error2, deferred, retryat*

message

is descriptive text associated with the report. The text is terminated by a linefeed.

Any other format (as might be produced by subprocesses) is passed to standard output for logging in the scheduler log. The *retryat* response will assume the first word of the text is a numeric parameter, either an incremental time in seconds if prefixed by “+”, or otherwise an absolute time in seconds since UNIX epoch.

The exit status is a code from `<sysexits.h>`.

22.7. Canned (Error) Message Files

FIXME! TEXT TO BE INSERTED HERE.

22.8. Security Issues

FIXME! TEXT TO BE INSERTED HERE.

Notes

1. The maximum number of supported TA's is actually probed by the **scheduler** at its startup; procedure is:

    1. Ask system for maximum number of file descriptors the scheduler's child can have.
    2. Substract "30" from resulting count.
    3. If child communication channel needs two file descriptors (no full-duplex pipe, e.g. `socketpair()` available), divide the leftover count by two.
    4. Result is the maximum number of TAs which document elsewere refers as "1000".

Chapter 23. Transport Agents References

The delivery agent programs normally form the final stage of message delivery.

These programs vary in function and facilities based on what they are doing to the messages, and what kind of channels they handle.

23.1. mailbox

```
- All options
- Internal Logic
- Tuning issues
- Customizability
- Logging ? (or move that to ADM?)
```

The **mailbox** is a ZMailer transport agent which is usually only run by the **scheduler**(8) program to deliver mail to local user mailbox files. The **mailbox** program must be run with root privileges and invoked with the same current directory as the **scheduler**, namely: `$POSTOFFICE/transport/`.

Recipient addresses are processed as follows:

- Strip doublequotes around the address, if any.
- Strip prefixing backslashes, if any.
- If the address starts with a "|", the rest of the recipient address string is interpreted as a shell command to be run.
- If the address starts with a "/", the recipient address is a filename to append the message to.
- Otherwise the recipient address must be a local user id.
- If user is not found, and the first character of the address is a capital letter, the entire address is folded to lowercase and the user lookup is retried.

If delivering to a user mailbox (`$MAILBOX/userid`) which does not exist, **mailbox** will try to create it. If the `$MAILBOX/` directory is mounted from a remote system this will succeed if the directory is group writable.

Some sanity checks are done on deliveries to files and mailboxes:

- The file being delivered to must have one link only, and must be either "`/dev/null`" or a regular file.
- The file lock must be held. (See below for a chapter about locks.)

There is a further sanity check on mailbox deliveries, namely if the mailbox is not empty the **mailbox** program will enforce 2 newlines as a separator before the message to be delivered. This guarantees that User Agents, like **Mail(1)**, can find the about-to-be delivered message even if the current contents of the mailbox is corrupt.

When delivering to a process (by starting a Bourne shell to execute a specified command line), the environment is set up to contain $PATH, $SHELL, $HOME, $USER, $SENDER, $UID environment variables. The $HOME and $USER values are the recipient user's home directory and login id respectively. The $SENDER value is the sender address for the message (as it would appear in a "`From `"-line), and the UID value is the owner id of the process. The `SIGINT` and `SIGHUP` signals are ignored, but `SIGTERM` is treated normally. If the process dumps core, it will be retried later. Otherwise any non-zero exit status is taken as a permanent failure, and will result in an error message back to the sender. The actual data delivered to a file, mailbox, or process are identical. It consists of the concationation of a UUCP style separator line, the message header specified in the message control file, and the message body from the original message file. The separator line starts with "`From `" and is followed by the sender address and a timestamp.

After all deliveries and just before exiting, the mailbox process will poke comsat(8C) in case recipients have turned on biff(1). The program may be compiled to look in the rwho files on the system for recipient names logged onto neighbouring hosts, in which case the comsat on the remote host will be poked. Even if this compile-time option is enabled, this will only be done for users that have a "`.rbiff`" file in their home directory. (Unless an "`-DRBIFF_ALWAYS`" compile option is used.)

Usage:

mailbox [-8] [-M] [-c *`channel`*] [-h *`localpart`*] [-l *`logfile`*] [-VabrH]

Parameters:

`-c "channel"`

specifies which channel name should be keyed on. The default is "local".

`-h "localpart"`

specifies which of the possible multiple recipients is to be picked this time. Default is "none", which selects all local channel recipients, however when the routing is done with scripts storing some tokens (other than "-") into the "host"-part, it is possible to process "host-wise", i.e. so that each user has his or her own lock-state, and not just everybody hang on the same lock(s)...

`-l "logfile"`

specifies a logfile. Each entry is a line containing message id, pre-existing mailbox size in bytes, number of bytes appended, and the file name or command line delivered to.

`-V`

prints a version message and exits.

`-a`

the access time on mailbox files is, by default, preserved across delivery, so that programs such as **login(1)** can determine, if new mail has arrived. This option disables the above action.

`-b`

disables biff notification.

`-r`

disables remote biff notification (if supported).

`-8`

enables the MIME-QP-decoder to decode incoming MIME-email with Quoted-Printable encoded characters.

`-M`

enables the creation of MMDF-style mail-folder in the incoming mail folder. The default is "classic" UNIX-style folder.

Interface:

As with all transport agents, the program reads relative pathnames of message control files from standard input (terminated with two linefeeds), and produces diagnostic output on the standard output. Normal diagnostic output is of the form:

```
id/offset<TAB>notify-data<TAB>status message
```

where id is the inode number of the message file, offset is a byte offset within its control file where the address being reported on is kept, status is one of ok, error, or deferred, and the message is descriptive text associated with the report. The text is terminated by a linefeed. Any other format (as might be produced by subprocesses) is passed to standard output for logging in the scheduler log. The exit status is a code from `<sysexits.h>`.

Locks:

The locking scheme used on the system is configurable at the runtime, and has separate parameters for mailboxes and files. The data is configurable with zenv variable `$MBOXLOCKS` in which the following characters have the meanings:

`:`

Separates mailbox locks, and file-locks at the string. The left side has mailbox locks, and the right side has locks for other regular files. (Files with explicit paths defined.)

`.`

For mailboxes only: Does "dotlock" (userid.lock), or (on Sun Solaris) the `maillock()` mechanism.

`F`

If the system has `flock()` system call, uses it to lock the entire file. (Ignored on systemswithout `flock()`)

`L`

If the system has `lockf()` system call, uses it to lock the entire file. (Ignored on systems without `lockf()`)

Locks are acquired in the same order as the key characters are listed.

The default for `lockf()` capable systems is: `MBOXLOCKS=".L:L"`.

You can choose insane combinations of lock mechanisms, which on some systems cause locks to fail always, like on *Linux-2.0* series where programs must not use both `lockf()` and `flock()` locks. It is extremely important that selected locking methods are consistent throughout the system with all programs trying to acquire locks on mail spools.

Environment:

The default location for user mailbox files is currently `/var/mail/`. This may be modified by setting the variable `$MAILBOX` in `/etc/zmailer.conf` to the directory containing user mailbox files, for example `/usr/spool/mail/`. This is best done in the ZMailer Config file. The variable `$MBOXLOCKS` is used to define locking schemes used for mailbox spool files, and separately for other regular files.

Security:

Like all parts of ZMailer, the mailbox chooses to err on the overly cautious side. In thecase of pipes being run under the mailbox, the program in the pipe is started through `/bin/sh` with severely sanitized environment variables, and with only the file descriptors `STDIN`, `STDOUT`, and `STDERR`. Programs are refused from running, if address analysis has found suspicuous data; external messages cannot directly run programs, nor those addresses that have had a security breach detected during `~/.forward`- or other aliasing analysis. (Same applies also with writing into explicitely named files.)

The pipe subprogram is run with user-id it gets thru the address privilege analysis during message routing, and it gets the groupid through lookup of `getpwuid(uid)`. That is, if you have multiple usernames with same uid, there are no guarantees as to which of them is used for the gid entry.

Subprogram Envonmrm´e:

The mailbox sets the following eight environment variables for the subprograms it runs in the pipes:

HOME

The home directory path is taken from abovementioned `getpwuid()` lookup.

USER

Likewise the textual username.

SENDER

is the incoming “MAIL FROM:<..>” address without brackets. For an incoming error message, value “<>” is used.

ORCPT

when present, is the XTEXT encoded ORCPT value received at the message injection into this system. See RFC 1891 for details.

ENVID

when present, is the XTEXT encoded ENVID value received at the message injection into this system. See RFC 1891 for details.

ZCONFIG

is the location of the ZMailer ZENV file.

MAILBIN

is the value from ZENV.

MAILSHARE

is the value from ZENV.

23.2. hold

```
- All options
- Internal Logic
- Tuning issues
- Logging ? (or move that to ADM?)
```

hold - zmailer deferred processing transport agent

Description:

hold is a ZMailer transport agent which is usually only run by the **scheduler(8)** program to test conditions for reprocessing of previously deferred message addresses.

The **hold** program must be run with the same current directory as the **scheduler**, namely: `$POSTOFFICE/transport/`.

The program will interpret the host part of an address destined for its channel as a condition that must be met before the original address (in the user part) can be reprocessed by the **router**. The condition specification contains a general condition class name followed by colon followed by a parameter string. The currently supported condition classes are:

`ns`

succeeds when the nameserver lookup indicated by the parameter does not produce a temporary nameserver error. The parameter is a domain name followed by a slash followed by a standard Internet nameserver Resource Record type name.

`timeout`

succeeds when the time given by the parameter (in normal seconds-since-epoch format) has passed.

`io`

succeeds 10% of the time, to allow retry of temporary I/O failures.

`script`

runs the named shell script with the optional given argument. The parameter is a simple name, the shell script name within the `$MAILBIN/bin/` directory, optionally followed by a slash followed by an argument to be passed to the shell script.

For example:

```
NS:nic.ddn.mil/cname
```

```
TIMEOUT:649901432
IO:error
SCRIPT:homedir/joe
```

The condition class name is case-insensitive but is capitalised by convention. The parameter strings are case-preserved for condition class-specific interpretation. Whitespace is not permitted.

The envelope of the resubmitted message is created from the sender and (no longer deferred) recipient addresses, and a “via suspension” header.

Description:

{\bf Usage} \begin{verbatim} hold [-c channel] [-V] \end{verbatim}

Description:

{\bf Parameters} {\tt -c channel} specifies which channel name should be keyed on. The default is hold. {\tt -V} prints a version message and exits.

Interface:

As all transport agents, the program reads relative path-names of message control files from standard input (terminated with two linefeeds), and produces diagnostic output on the standard output. Normal diagnostic output is of the form:

```
id/offset/status message
```

where id is the inode number of the message file, offset is a byte offset within its control file where the address being reported on is kept, status is one of ok, error, or deferred, and the message is descriptive text associated with the report. The text is terminated by a linefeed. Any other format (as might be produced by subprocesses) is passed to standard output for logging in the scheduler log.

The exit status is a code from `<sysexits.h>`;.

23.3. smtp

```
- All options
- Internal Logic at conversions
- SMTP vs. LMTP
- Tuning issues
- Logging ? (or move that to ADM?)
```

smtp - zmailer SMTP client transport agent

smtp is a ZMailer transport agent which is usually only run by the **scheduler(8)** to transfer messages to a remote Internet host using the SMTP protocol. The **smtp** program must be run with the same current directory as the **scheduler**, namely `$POSTOFFICE/transport/`.

The program scans the message control files named on `STDIN` for addresses destined for its channel and the host given on the command line. If any are found, all matching addresses and messages are transferred in a single SMTP conversation. The destination host might in fact be served by any available mail exchanger for that host.

Usage:

```
smtp [ -78deEHrPsVxW ] [ -c channel ] [ -h heloname ] [ -l logfile ]
[ -p remote-port ] [ -T timeout ] [ -F forcedest] [ -L localidentity ] host
```

Parameters:

`-7`

forces SMTP channel to be 7-bit, and thus forcing all 8-bit texts to be MIME-QP-encoded for the transport.

`-8`

forces SMTP channel to be 8-bit-clean, and as such, to decode the message while transporting it (is it is MIME QP encoded).

`-c channel`

specifies which channel name should be keyed on. The default is smtp.

`-d`

turns on debugging output.

`-e`

asks that for every destination address specification with a matching channel name, an MX lookup is done on the hostname to see whether the currently connected host can provide service for that destination. The default is to just do a textual name comparison with the destination hostname as given on the command line.

`-e`

use the “EHLO”-greeting only if the remote server initial banner reports “ESMTP” on it.

`-h host`

specifies the hostname for the SMTP HELO greeting. The default is the hostname of the local system, as returned by `gethostname(2)` or `uname(2)`.

`-F forcedest`

overrides delivery destination by forcing all email to be sent to given forcedest IP-number/hostname.

`-H`

Disable the per default active forced 8-bit headers conversion into MIME-2-format.

`-L localident`

specifies (for multi-homed machines) that they should use specified identity when connecting to the destination. Think of server with multiple IP numbers due to virtual hosting, for example. At such systems there may be situation when virtual identity needs to be used for reaching the destination system.

`-l logfile`

specifies a log file where the complete SMTP command transaction will be copied. Each line in the log will be prefixed with the process id of the transport agent process, so the same log file can be used by all SMTP clients.

`-r`

asks to set up SMTP connections using a source TCP port number under 1024. This is in the range of port numbers only available to a privileged process on some UNIX systems, which has led to some misguided attempts at mail security based on this mechanism.

`-s`

asks to report the progress of the SMTP conversation and data transfer on the command line in a way that will be visible to **ps(1)**.

`-x`

turns off MX lookups on delivery connections. This may be used ignore public MX knowledge and do exactly what the **router** says in cases where delivering to an explicit IP address is inappropriate.

`-P`

disable SMTP-PIPELINING usage (ESMTP keyword: PIPELINING)

`-T timeout`

specifies the timeout, in seconds, when waiting for a response to an SMTP command. The timeout applies to all SMTP command-response exchanges except for the acknowledgement after terminating the DATA portion of a message transaction (after sending the "." CRLF sequence). The default timeout is 10 minutes, the minimum acceptable value is 5 seconds. The timeout on the DATA acknowledgement is large, at least 10 minutes.

`-V`

prints a version message and exits.

`-W`

turns on the DNS WKS checking, and if the remote system does not have SMTP in its WKS-bits, email delivery to such address is aborted with an error message.

Interface:

As all transport agents, the program reads relative path names of message control files from standard input (terminated with two linefeeds), and produces diagnostic output on the standard output. Normal diagnostic output is of the form:

```
id/offset<TAB>notify-data<TAB>status message
```

where id is the inode number of the message file, offset is a byte offset within its control file where the address being reported on is kept, status is one of ok, error, or deferred, and the message is descriptive text associated with the report. The text is terminated by a linefeed. Any other format (as might be produced by subprocesses) is passed to standard output for logging in the scheduler log.

The exit status is a code from `<sysexits.h>`.

Extended SMTP:

When a user sends out 8-bit mail with the proper headers, this module can send it out to conforming servers either in 8-bit transparent manner, or down-converting "Content-Transfer-Encoding: 8BIT" to "Content-Transfer-Encoding: 7BIT" or to "Content-Transfer-Encoding: QUOTED-PRINTABLE" depending on what is the mail contents. This works only with "Content-Type: TEXT/PLAIN" thus no fancy MULTIPART/ALTERNATE et.al. schemes.. When "Content-Transfer-Encoding:"-header is not present in the headers, and recipient has not declared 8-bit SMTP capability, mail contents are treated with old 7-bit stripping method.

23.4. sm - zmailer Sendmail compatible transport agent

```
- ALL options, comparison against sendmail M-flags
- Internal Logic (incl. conversions)
- Tuning issues
- Logging ? (or move that to ADM?)
```

sm is a ZMailer transport agent which is usually only run by the **scheduler(8)**, to deliver messages by invoking a program with facilities and in a way compatible with a sendmail MTA. The **sm** program must be run with the same current directory as the **scheduler**, namely `$POSTOFFICE/transport/`.

The program scans the message control files named on `STDIN` for addresses destined for the channel and/or the host given on the command line. If any are found, all matching addresses and messages are processed according to the specifications for the mailer in the configuration file.

The exit status of a mailer should be one of the standard values specified in `<sysexits.h>`. Of these, `EX_OK` indicates successful delivery, and `EX_DATAERR`, `EX_NOUSER`, `EX_NOHOST`, `EX_UNAVAILABLE`, and `EX_NOPERM` indicate permanent failure. All other exit codes will be treated as a temporary failure and the delivery will be retried.

Usage:

sm [-8] [-H] [-Q] [-V] [-f *`configfile`*] -c *`channel`* -h *`host mailer`*

Parameters:

`-8`

tells that the output is 8-bit clean, and for any MIME message with QUOTED-PRINTABLE encoding the coding can be decoded.

`-Q`

tells that the transport channel will likely treat poorly control characters like TAB, and possibly SPACE too.. This encodes them all by using QUOTED-PRINTABLE encoding.

`-f configfile`

specifies the name of a configuration file containing specifications of the various known sendmail compatible mailer programs: how to invoke them and how to process messages for them. The default is `$MAILSHARE/sm.cf`.

`-c channel`

specifies which channel name should be keyed on. There is no default. If this option is not specified, the `-h` option must be.

`-h host`

specifies which host name should be keyed on. There is no default. If this option is not specified, the `-c` option must be.

`-h host`

prints a version message and exits.

23.4.1. configuration of sm

sm is a ZMailer's **sendmail**(8) compatible **transport agent** to deliver messages by invoking a program with facilities and in a way compatible with a **sendmail**(8) MTA.

The program scans the message control files named on stdin for addresses destined for the channel and/or the host given on the command line. If any are found, all matching addresses and messages are processed according to the specifications for the mailer in the configuration file.

The exit status of a mailer should be one of the standard values specified in *#include <`sysexits.h`>*. Of these, EX_OK indicates successful deliver, and EX_DATAERR, EX_NOUSER, EX_NOHOST, EX_UNAVAILABLE, and EX_NOPERM indicate permanent failure. All other exit codes will be treated as a temporary failure and the delivery will be retried.

Usage:

sm [-8] [-H] [-Q] [-V] [-f *configfile*] -c *channel* -h *host mailer*

Configuration:

The configuration file `$MAILSHARE/sm.conf` associates the mailer keyword from the command line with a specification of a delivery program. This is very similar to the way the definition of a "mailer" in **sendmail**(8). It requires flags, a program name, and a command line specification. These are in fact the fields of the entries of the configuration file. Lines starting with whitespace or a "#" are ignored, and all others are assumed to follow format shown in figure Figure 23-1.

Figure 23-1. Sample `sm.conf` file

```
#
# M          F =     P =                            A =
# the following entries are in active use at this site:
uucp        U   /usr/bin/uux          uux - -r -a$g -gC $h!rmail ($u)
usenet      m   ${MAILBIN}/ta/usenet  usenet $u
#
bitbucket   -  @MAILBIN@/ta/bitbucket  bitbucket
#
```

```
#
# bitnet stuff F=hu not set?
#
bsmtp3      snmSX /usr/local/funetnje/bmail bmail -b $h $g $u
bsmtp3rfc   snmSX /usr/local/funetnje/bmail bmail -b $h $g $u
bsmtp3nd    snmSX /usr/local/funetnje/bmail bmail -nd $h $g $u
bsmtp3ndrfc snmSX /usr/local/funetnje/bmail bmail -nd $h $g $u
defrt1      snS   /usr/local/funetnje/bmail bmail $g $u
bitnet2     snS   /usr/local/funetnje/bmail bmail $g $u
#
# the following entries are included to illustrate other possibilities
#
#local  mS  /usr/lib/mail/localm           localm -r $g $u
cyrus   Pn  /usr/cyrus/bin/deliver         deliver -e -m $h -- $u
#           # CYRUS example from: Tom Samplonius <tom@sdf.com>
procm sSPfn @PROCMAIL@      procmail -a $h -d $u
#           # Procmail example from: Ken Pizzini <ken@spry.com>
#
#prog  -    /bin/sh                        sh -c $u
#tty   rs   /usr/local/to                  to $u
#ean   mn   /local/lib/ean/mailer          mailer -d $u
#test  n    /local/lib/mail/bin/test       test $u
#
```

The mailer field extends from the beginning of the line to the first whitespace. It is used simply as a key index to the configuration file contents. One or more whitespace is used as the field separator for all the fields.

The flags field contains a concatenation of one-letter flags. If no flags are desired, a “-” character should be used to indicate presence of the field. All normal sendmail (*of 5.x era..*) flags are recognized, but the ones that do not make sense in the context of ZMailer will produce an error (or some are ignored). The flags that change the behaviour of **sm** are:

`b`

will activate BSMTP-type wrapping with a “hidden-dot” algorithm; e.g. quite ordinary SMTP stream, but in “batch mode”.

`B`

The first “`B`” turns on similar BSMTP wrapping as “`b`”, but adds SIZE and, if the **sm** is started with option “`-8`”, also 8BITMIME options. The second “`B`” adds there also DSN (Delivery Status Notification) parameters.

`E`

will prepend “>” to any message body line starting with “*From* ”. (Read: “From-space”)

`f`

adds “`-f sender`” arguments to the delivery program.

`n`

will not prepend a “*From* ”-line (normal mailbox separator line) to the message.

`r`

adds “`-r sender`” arguments to the delivery program.

`S`

will run the delivery program with the same real and effective uid as the **sm** process. If this flag is not set, the delivery program will be run with the real uid of the **sm** process. This may be useful if **sm** is setuid.

`m`

informs **sm** that each instance of the delivery program can deliver to many destinations. This affects `$u` expansion in the argument list, see below.

`P`

prepends a "Return-Path:" header to the message.

`U`

will prepend a "*From* "-line, with a "remote from myuucpname" at the end, to the message. This is what is expected by remote **rmail**(1) programs for incoming UUCP mail.

`R`

use CRLF sequence as end-of-line sequence. Without it, will use LF-only end-of-line sequence.

`X`

does SMTP-like "hidden-dot" algorithm of doubling all dots that are at the start of the line.

`7`

will strip (set to 0) the 8th bit of every character in the message.

The path field specifies the location of the delivery program. Relative pathnames are allowed and are relative to the `$MAILBIN/` directory.

The arguments field extends to the end of the line. It contains whitespace separated `argv` parameters which may contain one of the following sequences:

`$g`

which is replaced by the sender address.

`$h`

which is replaced by the destination host.

`$u`

which is replaced by the recipient address. If the "`m`" mailer flag is set and there are several recipients for this message, the argument containing the "`$u`" will be replicated as necessary for each recipient.

23.5. expirer

`- All options`

```
- Internal Logic
- Tuning issues
- Logging ? (or move that to ADM?)
```

FIXME! FIMXE! write me.. (about the tool to kill out messages from the queue)

23.6. libta - Transport Agent Support Library

This is the library that all transport agents use, and several of its functions are intended to aid message processing.

23.6.1. Function groupings

Transport agent support library function groups are:

- Message file manipulation routines.
- Diagnostics routines.

23.6.2. Function listings

Text to be inserted here.

23.6.3. Function usage examples

Text to be inserted here.

23.7. Security Issues

Text to be inserted here.

Chapter 24. ZMailer Utilities Reference

There is considerable collection of various utilities in the ZMailer sources. Not all of them even become installed into your system in all situations.

24.1. zmailer command script

The **zmailer** command script is a wrapper for driving various sub-utilities, and in some cases, honouring flags like “freeze-state”, which administrator may set to keep system down over reboots while some maintenance acitivity is under way.

Plain “**zmailer**” command is synonymous to “**zmailer start**”.

24.1.1. zmailer bootclean

This removes all internal process reference PID files from `$POSTOFFICE/` directory.

Highly recommended for your system startup scripts before starting servers

24.1.2. zmailer start

Without further parameters this starts the entire ZMailer system by starting subservers: **smtpserver**, **router**, and **scheduler**.

Giving parameter (one or more of subsystem names above) (re)starts just that (or those) subsystem(s).

If the system is in “frozen” state, start fails. See “**zmailer freeze**” below.

24.1.3. zmailer stop, zmailer kill

Without further parameters, this terminates the main daemons of the ZMailer (**smtpserver**, **router**, and **scheduler**) by sending `SIGTERM` to them.

Giving parameter (one or more of subsystem names above) stops just that (or those) subsystem(s).

24.1.4. zmailer nuke

Without further parameters, this kills the main daemons of the ZMailer. (**smtpserver**, **router**, and **scheduler**) by sending `SIGKILL` to them.

Giving parameter (one or more of subsystem names above) stops just that (or those) subsystem(s).

24.1.5. zmailer router

Synonym to “**zmailer start router**”, (re)starts the **router** process(es).

24.1.6. zmailer scheduler

Synonym to "**zmailer start router**", (re)starts the **scheduler** process.

24.1.7. zmailer smtp(server)

Synonym to "**zmailer start smtpserver**", (re)starts the **smtpserver** process.

24.1.8. zmailer newdb

A complicated subsystem on its own merits used to re-generate **router** configuration data for various database lookups.

This runs utility called "***newdbprocessor***" with its only argument of "`$MAILVAR/db/dbases.conf`".

24.1.9. zmailer newal(iases)

Re-generates "`aliases`" database, wrapper of "**newaliases**", and superceded by "***zmailer newdb***".

24.1.10. zmailer newf(qdnaliases)

Re-generates "`fqdnaliases`" database, wrapper of "**newfqdnaliases**", and superceded by **zmailer newdb**.

24.1.11. zmailer new-route(s)

Compiles "`routes`" database, and superceded by **zmailer newdb**.

24.1.12. zmailer new-local(names)

Compiles "`localnames`" database, and superceded by **zmailer newdb**.

24.1.13. zmailer logsync

A special command sending signals to subsystems needing them for reopening their possibly long-living logfile opens.

To be used *after* other methods have rotated the logfiles to new names, but before anything further is done to them.

These days only the **scheduler** needs it, and if you have a choice, use this only after the "scheduler" logfile is rotated.

24.1.14. zmailer logrotate

Runs ZMailer sub-utility “rotate-logs.sh”.

24.1.15. zmailer resubmit

Moves files from `$POSTOFFICE/deferred/` to main **router** input directory (`$POSTOFFICE/router/`.)

24.1.16. zmailer cleanup

Run this from your root crontab!

This cleans from `$POSTOFFICE/public/` files that are older than 2 days (48 hours), and from `$POSTOFFICE/postman/` files with names starting with a digit and aged over 7 days.

24.1.17. zmailer freeze

This sets a flag which is honoured by subsystem start functions:

```
# /opt/mail/bin/zmailer freeze
freeze
# /opt/mail/bin/zmailer router
router Sorry, ZMailer is frozen, won't start anything until thawed !
* CHECK THAT THE FREEZE CONDITION ISN'T DUE TO E.G. MAINTENANCE *
```

24.1.18. zmailer thaw, zmailer unfr(eeze)

Thaws the previously frozen ZMailer system so that **zmailer start** will be able to start subsystems.

24.2. The newdbprocessor script

FIXME! WRITEME! (See Section E.4.1 for the configuration file)

```
- Input file syntax
- Supported database types, and what is done to them
- Additional notes ?
```

24.3. The newdb script

This is elementary wrapper script building binary databases with **makedb** utility into a temporary file, and replacing the old files with the new ones in proper order for the **router**’s automatic source change detecting relation parameter `-m` to work correctly.

newdb [`-u`|`-l`] [`-a`] [`-s`] [`-t` *`dbtype`*] *`/db/path/basename`* [*`input-file-name`*]

This script uses system `ZCONFIG` file to find out the desired database type, and derives the actual database file names from the variable.

Suffix selection rules are:

```
dbm     .pag and .dir
ndbm    .pag and .dir
gdbm    .gdbm
btree   .db
bhash   .dbh
```

24.4. The makedb utility

This utility is used by the ZMailer system to compile source files to various binary databases.

The way the ZMailer uses DBM entries is by using C-strings with their terminating `NUL` included at keys, and at data.. Thus the length is `strlen(string)+1`, not `strlen(string)` !

```
WARNING: Policy data parsing does use unchecked buffers!

\begin{verbatim}
Usage: makedb [-apsvluA] dbtype database.name [infilename|-]
\end{verbatim}

-a: aliasinput
-p: policyinput
-A: append-mode
-l: pre-lowercasify the key
-u: pre-uppercasify the key
-s: be silent
-v: be verbose

Dbtypes are: {\tt ndbm gdbm btree bhash}

If no {\tt infilename} is defined, {\tt database.name} is assumed.

\begin{description}
\item[{\tt NDBM}] \mbox{}

appends {\tt .pag}, and {\tt .dir}
into the actual db file names.

\item[{\tt GDBM}] \mbox{}

{\bf does not} append {\tt .gdbm}
into the actual db file name.

\item[{\tt BTREE}] \mbox{}

{\bf does not} append {\tt .db}
into the actual db file name.
```

```
\item[{\tt BHASH}] \mbox{}

appends {\tt .pag}, and {\tt .dir}
into the actual db file names.

\end{description}


The {\tt -a} option is for parsing input that comes in
{\tt aliases} format: {\tt key: data,in,single,long,line}
```

24.5. The dblook utility

```
The way the ZMailer uses DBM entries is by using strings with
their terminating {\tt NULL} as keys, and as data.. Thus the
length is {\tt strlen(string)+1}, not {\tt strlen(string)} !

\begin{verbatim}
Usage: dblook [-dump] dbtype database.name [key]
\end{verbatim}


Dbtypes are: {\tt ndbm gdbm btree bhash}

\begin{description}
\item[{\tt NDBM}] \mbox{}

appends {\tt .pag}, and {\tt .dir}
into the actual db file names.

\item[{\tt GDBM}] \mbox{}

{\bf does not} append {\tt .gdbm}
into the actual db file name.

\item[{\tt BTREE}] \mbox{}

{\bf does not} append {\tt .db}
into the actual db file name.

\item[{\tt BHASH}] \mbox{}

appends {\tt .pag}, and {\tt .dir}
into the actual db file names.

\end{description}

%\end{multicols}


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\clearpage
```

24.6. The policy-builder.sh script

```
#! /bin/sh
#
# Sample smtp-policy-db builder script.
#
# This merges following files from $MAILVAR/db/ directory:
#       smtp-policy.src
#       localnames                 ('= _localnames')
#       smtp-policy.relay.manual ('= _full_rights')
#       smtp-policy.relay        ('= _full_rights')
#       smtp-policy.mx.manual    ('= _relaytarget')
#       smtp-policy.mx           ('= _relaytarget')
#       smtp-policy.spam         ('= _bulk_mail')
#       smtp-policy.spam.manual  ('= _bulk_mail')
#
# These all together are used to produce files:  smtp-policy.$DBEXT
# The produced database retains the first instance of any given key.
#

#FLAG=
#while getopts n c; do
#  case $c in
#    n)       FLAG=$c;;
#    ?)       exit 2;;
#  esac
#done
#shift `expr $OPTIND - 1`

ZCONFIG=@ZMAILERCFGFILE@
. $ZCONFIG

DBDIR="$MAILVAR/db/"
USAGE="Usage: $0 [-n] [-d dbdir]"

while [ "$1" != "" ]; do
    case "$1" in
        -n)
            FLAG=n
            ;;
        -d)
            shift
            DBDIR=$1
            if [ ! -d $DBDIR ]; then
                echo $USAGE
                exit 1
            fi
            ;;
        ?)
            echo $USAGE
            exit 0
            ;;
        *)
            echo $USAGE
            exit 2
            ;;
    esac
    shift
done


umask 022

cd $DBDIR

if [ ! -f smtp-policy.src ] ; then
        echo "No $DBDIR/smtp-policy.src input file"
        exit 64 # EX_USAGE
fi

# -- Former '-f' flag data (non)retrieval section removed

# Fork off a subshell to do it all...
(
  # The basic boilerplate
  cat smtp-policy.src

  # Localnames
  echo "# ----------"
  echo "# localnames:"
  cat localnames | \
  awk '/^#/{next;} NF >= 1 {printf "%s = _localnames\n",$1;}'

  # smtp-policy.relay
  # (Lists NETWORKS (NO DOMAINS!) that are allowed to use us as relay)
  # (well, actually it could also list e.g.: ".our.domain" if it would
  #  be fine to allow relaying from anybody whose IP address reverses to
  #  domain suffix ".our.domain")
  if [ -f smtp-policy.relay.manual ] ; then
    echo "# -------------------------"
    echo "# smtp-policy.relay.manual:"
    cat smtp-policy.relay.manual | \
    awk '/^#/{next;}
        {printf "%s = _full_rights\n",$0;next;}'
  fi
  if [ -f smtp-policy.relay ] ; then
    echo "# ------------------"
    echo "# smtp-policy.relay:"
    cat smtp-policy.relay | \
    awk '/^#/{next;}
        {printf "%s = _full_rights\n",$0;next;}'
  fi

  # smtp-policy.mx.manual
  # (Lists domains that are allowed to use us as inbound MX relay for them)
  if [ -f smtp-policy.mx.manual ] ; then
    echo "# ----------------------"
    echo "# smtp-policy.mx.manual:"
    cat smtp-policy.mx.manual | \
    awk '/^#/{next;} NF >= 1 {printf "%s = _relaytarget\n",$0;}'
  fi
  # smtp-policy.mx
  # (Lists domains that are allowed to use us as inbound MX relay for them)
  if [ -f smtp-policy.mx ] ; then
```

```
      echo "# ---------------"
      echo "# smtp-policy.mx:"
      cat smtp-policy.mx | \
      awk '/^#/{next;} NF >= 1 {printf "%s = _relaytarget\n",$0;}'
    fi

    # smtp-policy.spam
    # (Lists users, and domains that are known spam sources)
    # (We use file from "http://www.webeasy.com:8080/spam/spam_download_table"
    #  which is intended for QMAIL, and thus needs to be edited..)
    if [ -f smtp-policy.spam -o -f smtp-policy.spam.manual ] ; then
      echo "# ---------------------------"
      echo "# smtp-policy.spam{,.manual}:"
      ( if [ -f smtp-policy.spam ] ; then
          cat smtp-policy.spam
        fi
        if [ -f smtp-policy.spam.manual ] ; then
          cat smtp-policy.spam.manual
        fi ) | tr "[A-Z]" "[a-z]" | sed 's/^@//g' | sort | uniq | \
      awk '/^\[/{ # an address block to reject
              printf "%s  rejectnet +\n",$0;
              next;
          }
          NF > 0 { # All other cases are usernames with their domains
              printf "%s  = _bulk_mail\n",$0;
          }'
    fi

  # --------- end of subshell
  ) > smtp-policy.dat

  umask 022 # Make sure the resulting db file(s) are readable by all

  # Build the actual binary policy database (-p), and if the input
  # has same key repeating, append latter data instances to the first
  # one (-A):

  $MAILBIN/makedb -A -p $DBTYPE smtp-policy-new smtp-policy.dat || exit $?

  case $DBTYPE in
  dbm)
          mv smtp-policy-new.dir  smtp-policy.dir
          mv smtp-policy-new.pag  smtp-policy.pag
          ;;
  ndbm)
          mv smtp-policy-new.dir  smtp-policy.dir
          mv smtp-policy-new.pag  smtp-policy.pag
          ;;
  gdbm)
          mv smtp-policy-new.gdbm smtp-policy.gdbm
          ;;
  btree)
          mv smtp-policy-new.db   smtp-policy.db
          ;;
  esac

  exit 0
```

24.7. autoanswer

The **autoanswer** program is intended to be placed into system global aliases database as following entry:

```
  autoanswer:  "| /path/to/MAILBIN/autoanswer"
```

It yields a reply message for all, except the error messages, nor to those with `X-autoanswer-loop:` header in them.

The reply sends back the original incoming message headers in the message body along with some commentary texts.

The program is, in reality, a perl script which can easily be tuned to local needs.

```
#!@PERL@

###########################################################################
#
# Autoanswer.pl 1.0 for ZMailer 2.99.48+
# (C) 1997 Telecom Finland
#          Valtteri Karu <valtteri.karu@tele.fi>
#
# This program sends autoreply and the original headers to the originator
# of the message. Version 2.99.48+ of the Zmailer is required for detecting
# possible false addresses.
#
# USAGE:
#
# Create an alias for the address want to use:
# autoreply: "|/path/to/autoanswer.pl"
#
###########################################################################

$nosend = 0;
$double = 0;
$address = $ENV{'SENDER'};

if( ! -r "$ENV{'ZCONFIG'}") {
    LOG("zmailer.conf missing");
    exit 2;
}

open(ZMAILER,"< $ENV{'ZCONFIG'}" );
while(<ZMAILER>) {
    chomp;
    split(/=/);
    $ZMAILER{$_[0]}=$_[1];
}

close ZMAILER;

$logfile = $ZMAILER{'LOGDIR'} . "/autoanswer";

while (<STDIN>) {

    $text = $_;
```

```
            if (($text eq "\n") && ( $double == 1)) {
         last;
            }

            if (($text eq "\n") && ( $double == 0)) {
         $double = 1;
         next;
            }

            if ($text =~ m/^X-autoanswer-loop:/i) {
         $nosend = 1;
         LOG("Looping message, sender=$address");
            }

            $double = 0;

            push(@header,$text);
        }

        if (($address eq '<>') || $nosend ) {
            LOG("SENDER invalid");
            exit 1;
        }


        $outfile = $ZMAILER{'POSTOFFICE'} . "/public/autoanswer.$$";
        #$outfile = "/tmp/aa.$$";
        $now = time;
        $txttime = localtime(time);

        open(OUT,">$outfile");
        select(OUT);
        print "channel error\n";
        print "to $address\n";
        print "env-end\n";
        print "From: Autoreply service <postmaster>\n";
        print "To: $address\n";
        print "Subject: Autoreply\n";
        print "X-autoanswer-loop: Megaloop \n\n";
        print "      This is autoreply answer message by your request.\n\n";
        print "      Original message was received at UNIX time $now;\n";
        print "      which means '$txttime' in cleartext.\n\n";
        print "      Headers were:\n\n";
        print "---------------------------------------------------------------------\n"
        print @header;
        print "---------------------------------------------------------------------\n"
        print "\n      Have a nice day.\n";
        select(STDOUT);
        close OUT;
        $inode=(stat($outfile))[1];
        $newfile=$ZMAILER{'POSTOFFICE'} . "/router/$inode";
        rename($outfile, $newfile);
        LOG("Sent to $address");
        exit 0;

        sub LOG {
```

```
 open(LOGf, ">>$logfile");
 $ttime = localtime(time);
 printf (LOGf "$ttime autoanswer: @_\n");
 close LOGf;
}
```

24.8. vacation

vacation automatically replies to incoming mail. The canned reply is contained in the file `~/.vacation.msg`, that you should create in your home directory (or the file **`Msgfile`** specified by the `-m` option).

This file should include a header with at least a “Subject:” line (it should not include a “To:” line — if you want, you may include “From:” line, especially if you use the `-m` option), for example.

Usage:

To start **vacation**, run the command **`vacation start`**. It will create a `~/.vacation.msg` file (if you don’t already have one) in your home directory containing the message you want to send people who send you mail, and a `~/.forward` file in your home directory containing a line of the form:

```
"\name", "|/opt/mail/bin/vacation name"
```

where name is your login name. Make sure these files and your home directory are readable by everyone. Also make sure that no one else can write to them, and that no one can write to your home directory. Like this:

```
chmod og-w $HOME $HOME/.forward
```

)

To stop vacation, run the command **vacation stop**. It will move the `~/.forward` file to `~/.vacforward`, and the automatic replies will stop.

vacation ’start’
vacation ’stop’
vacation `-I`
vacation [`-tN`] [`-m`*`Msgfile`*] [`-d`] [*`user`*]

Parameters:

`-I`, `-i`

initialize the `.vacation.pag`, and `.vacation.dir` files (or whatever the system supported database is), and start *vacation*.

If the `-I` (or `-i`) flag is not specified, *vacation* tries to reply to the sender.

`-tN`

Change the interval between repeat replies to the same sender. The default is one week. A trailing `s`, `m`, `h`, `d`, or `w` scales N to seconds, minutes, hours, days, or weeks respectively.

`-m`*`MsgFile`*

specifies the file in which the message to be sent is kept. The default is `$HOME/.vacation.msg`.

`-r`

interval defines interval in days when not to answer again to the same sender. (Default is 1 day.)

`-d`

disables the list of senders kept in the `.vacation.pag` and `.vacation.dir` files. (Or whatever database format is being used.)

Example:

```
Subject: I am on vacation
```

```
I am on vacation until July 22. If you have something urgent, please
contact Joe Jones (joe@blah.utoronto.ca). --john
```

No message is sent if the “user” specified in the vacation command (if nothing is specified, it uses your username) does not appear explicitly in the “To:” or “Cc:” lines of the message, which prevents messages from being sent back to mailing lists and causing loops.

A list of senders is kept in the files `~/.vacation.pag` and `~/.vacation.dir` in your home directory. These are dbm database files. (Note: not all database systems have two files, either may be missing.) The vacation message is in `~/.vacation.msg` and the automatic reply is activated by the `~/.forward` (and saved in `~/.vacforward`) The default vacation message is stored in `$MAILSHARE/vacation.msg`

On machines running ZMailer, the “name” argument to **vacation** is optional, and the $USER environment variable is used to determine where to look for the message and the list of previous recipients.

The `$SENDER` variable is checked first to determine the reply destination. It is normally set to the SMTP “MAIL FROM:” address or equivalent. This is an additional safeguard against sending replies to mailing lists, the PostMaster or the mailer daemon, since standards and common sense dictate that it never points back to an address that could cause a loop. The “From ” line is used only as a last resort.

V. Appendices

Appendix A. Sample Router Configuration Scripts

Text to be inserted here.

The following are examples of the router configuration scripts SMTP+UUCP.cf, crossbar.cf, process.cf, and rrouter.cf.

A.1. `SMTP+UUCP.cf`

ZMailer 2 configuration file for a generic SMTP host (with UUCP links)

```
ZCONFIG=@ZMAILERCFGFILE@

. $ZCONFIG

PATH=.:$MAILSHARE/cf:$MAILBIN/bin ; export PATH
PS1=z$PS1
```

Configure error logging (squirrel)

```
squirrel -breakin
squirrel badheader
```

Domains with these toplevels will not be canonicalized via DNS lookup. This list is from ISOC table of 16-April-95.

The quoted string {\tt "ad...zw"} should be on one line in the actual SMTP+UUCP.cf file.

```
toplevels="ad ae af ag ai al am an ao aq ar as at au aw az ba bb bd
           be bf bg bh bi bj bm bn bo br bs bt bv bw by bz ca cc cf
    cg ch ci ck cl cm cn co com cr cu cv cx cy cz de dj dk dm
    do dz ec edu ee eg eh es et fi fj fk fm fo fr ga gb gd ge
    gf gh gi gl gm gn gov gp gq gr gt gu gw gy hk hm hn hr ht
    hu id ie il in int io iq ir is it jm jo jp ke kg kh ki km
    kn kp kr kw ky kz la lb lc li lk lr ls lt lu lv ly ma mc
    md mg mh mil ml mm mn mo mp mq mr ms mt mu mv mw mx my mz
    na nc ne net nf ng ni nl no np nr nt nu nz om org pa pe pf
    pg ph pk pl pm pn pr pt pw py qa re ro ru rw sa sb sc sd se
    sg sh si sj sk sl sm sn so sr st sv sy sz tc td tf tg th tj
    tk tm tn to tp tr tt tv tw tz ua ug uk um us uy uz va vc ve
    vg vi vn vu wf ws ye yu za zm zr zw"
```

The transport preference order

```
protocols='routes smtp uucp'
```

Will the MAILVAR/lists/listname show out sender identity as either: owner-listname, or: listname-owner?

```
if true ; then # Change to "false" to get "pre-owner" mode
        preowner=""
        postowner="-owner"
else
        preowner="owner-"
        postowner=""
fi
```

Does our “local” channel accept domain (@) at the user part? ZMailer’s mailbox does accept. If you use something else, and it doesn’t accept, comment this away.

```
localdoesdomain=1
```

We may want .forward and mailing list files to be private, i.e., we ignore the current privileges when checking the privileges of such files. Don’t add ‘include’ to this list, since anyone can :include: any file.

```
private='.forward maillist'
```

Set up dependency checking.

```
. consist.cf
require siteinfo router crossbar process server
```

The following are standard setup files and must be loaded in this order

```
. standard.cf
. trusted.cf
```

Load the databases so they and the variables defined (e.g. network-specific node names for this host) can be used in the site specific configuration.

```
for method in $protocols
do
        test -f $MAILSHARE/cf/i-${method}.cf && . i-${method}.cf
done

mailconf () {
        local hname

        # My official hostname
        if [ -f /bin/hostname ]; then
                rawhostname=$(/bin/hostname)
        elif [ -f /etc/sys_id ]; then
                read rawhostname < /etc/sys_id
        else
                rawhostname=$(/bin/uname -n)
        fi

        hname=$(canon $rawhostname)
```

Try to discover the organizational domain name.

```
        orgdomain=$hname
        sift $hname in
        $rawhostname\.(.+)
                orgdomain=\1
                ;;
        tfis
        hostname=$hname

        # This is what it will say on out mail
        mydomain=$hostname
```

```
}

orgdomains=x
: ${MAILCONF:=/etc/mail.conf}
if [ ! -r $MAILCONF ]; then
        echo "$0: missing $MAILCONF: using the following values:"
        mailconf
        echo orgdomain=$orgdomain
        echo hostname=$hostname
        echo mydomain=$mydomain
        provide siteinfo
else
        . $MAILCONF && provide siteinfo
fi
[ "$orgdomains" = x ] && orgdomains=$orgdomain
```

Set hostname to enable message-id generation and checking.

```
hostname $hostname

. aliases.cf
. canon.cf
. rrouter.cf
. crossbar.cf

for method in $protocols
do
        . p-${method}.cf
done

. process.cf
. server.cf

consist || exit 1
```

A.2. `Crossbar.cf`

```
provide crossbar
```

The crossbar function makes the policy decisions of how the instance of a message between a particular sender and recipient should be treated. The 'from' and 'to' parameters are quads, i.e., in the form

(channel host user attributes)

The function may modify any of these elements of both the from and to addresses, and must select a message header address rewriting function to be applied to this message instance. If the return value is nil or empty, the instance is completely ignored, to the point that if there are no other recipients specified a complaint will be generated saying there are {\bf no} recipients specified.

```
crossbar (from, to) {
        local rewrite destination tmp
```

Count them... (in {\tt process.cf}) (we could use this as an ultimate duplicate remover too...)

```
        db add recipients "$(user $to)" "$(user $from)"
```

Intercept (drop, redirect, bounce, save) the message

```
        tmp=$(intercept "$(user $from)") &&
                case "$(car $tmp)" in
```

Dropping error types for from addresses is necessary to avoid mail loops.

```
                drop|error)
                        return ;;
                file)   LOGMSG="$LOGMSG $(car $(cdr $tmp))" ;;
                esac
```

Only intercept mail that is not from the local postmaster, so that error messages can find their way back.

```
        [ "$(channel $from)" = local -a "$(user $from)" = postmaster ] ||
        tmp=$(intercept "$(user $to)") &&
                case "$(car $tmp)" in
                drop)   return ;;
                error)  setf $(channel $to) error
                        setf $(host $to) $(car $(cdr $tmp))
                        ;;
                file)   LOGMSG="$LOGMSG $(car $(cdr $tmp))" ;;
                esac
```

If we do any alias expansion from the crossbar, we should do this:

```
db flush expansions
```

Determine which rewrite function (for message header addresses) to use.

```
        case $(channel $to) in
        smtp|smtpx)
                #case "$(channel $from)" in
                #smtp|smtpx)    # Address should be forwarded the way the arrive
                #       rewrite=null ;;
                #*)     rewrite=internet ;;
                #esac
                rewrite=internet
                ;;
        error)  rewrite=null ;;
        local)  case "$(channel $from)" in
                local)  #rewrite=intramachine
                        rewrite=internet ;;
                *)      # addresses should be saved the way they arrive
                        rewrite=null ;;
                esac
                ;;
        usenet) rewrite=internet ;;
        ean)    rewrite=ean_useratdomain ;;
        *)      # This is usually UUCP or BITNET
                # We want to determine the final destination host/domain
                destination="$(uucproute "$(user $to)")"
                if [ "$(host $to)" ]; then
                        destination="$(host $to)"!"$destination"
                fi
                sift "$destination" in
                .*!([^!]+)![^!]+
                        destination="\1" ;;     # destination domain
                .*\.(bitnet|netnorth|earn|cdn)
                        rewrite=smtp_useratdomain
                        break ;;                        # reply to user@domain
                .*      rewrite=internet ; break ;;     # default sensible thing
                tfis
                ;;
        esac
```

The alias expansion might want to modify the envelope sender of the message instance. Here we cooperate in the scheme which is to set the 'sender' attribute of the destination address.

```
        tmp="$(get $(attributes $to) sender)" && [ x"$tmp" != x ] &&
                from=(local "$tmp" "$tmp" $(attributes $from))


        case "$(channel $from)" in
        defrt1*)
                setf "$(user $from)" "$(bitnetroute "$(user $from)")"
                if [ $rewrite = internet ]; then
                        rewrite=bitnet2internet
                fi
                ;;
        esac
```

Rewrite the envelope addresses appropriately.

```
        case "$(channel $to)" in
#       uucp|local)
        uucp)
```

Local destination on a system that delivers in UCB Mail compatible mail spool files means that the From_ line must be in all-! form, which is the same as the UUCP transport requirement.

```
                setf "$(user $from)" "$(uucproute "$(user $from)")"
                setf "$(user $to)" "$(uucproute "$(user $to)")"
                sift "$(user $to)" in
                (.)!(.*)        if [ \1 = $(host $to) ]; then
                                        setf "$(user $to)" \2
                                fi
                                ;;
                tfis
                sift "$(user $to)" in
                (.)\.uucp!(.*)  setf "$(user $to)" \1!\2 ;;
                tfis
                ;;
#       smtp)
        smtp|smtpx|local|bsmtp3*)
                tmp="$(smtproute "$(user $from)")"
                sift "$tmp" in
                (@$hostname[:,].*)|([^@:,]+@$hostname)
                        break ;;
                .*
                        # tmp="@$hostname:$tmp"  # <-- that creates RFC-822
                                                 #     source-routing, AVOID!
                        tmp="$tmp"
                        ;;
                @(.+):(.+:.+)
```

```
                        tmp="@\1,\2" ; continue ;;
                tfis
                setf "$(user $from)" "$tmp"
                sift "$(user $to)" in
                (^/).*  setf "$(user $to)" "$(smtproute "$(user $to)")" ;;
                tfis

                ;;
        ean)
                setf $(user $from) "$(ean_useratdomain "$(user $from)")"
                setf "$(user $to)" "$(ean_useratdomain "$(user $to)")"
                ;;
        usenet)
                setf $(user $from) "$(uucproute "$(user $from)")"
                sift $(user $from) in
                $hostname!.*    ;;
                .*      setf $(user $from) $hostname!$(user $from) ;;
                tfis
                # newsgroup name only
                setf "$(user $to)" $(localpart "$(user $to)")
                ;;
#       bsmtp3|bsmtp3nd)
#               setf $(user $from) "$(bitnetroute "$(user $from)")"
#               tmp="$(bitnetroute "$(user $to)")"
#               sift "$tmp" in
#               .*@([^.]).uucp
#                       tmp="$(bitnetShortroute "$(user $to)")" ;;
#               tfis
#               setf "$(user $to)" "$tmp"
#               rewrite=bitnetShortroute
#               ;;

        defrt1)
                setf $(user $from) "$(bitnetroute "$(user $from)")"
                setf $(user $to) "$(bitnetroute "$(user $to)")"
                rewrite=bitnetroute
                sift "$(user $to)" in
                (.*)[!%](.+)@(.*)
                        to=(error bitnetgw "\3" $(attributes $to))
                        rewrite=null
                        ;;
                tfis
                ;;
        esac

        #log recipient: "$(channel $to)" "$(host $to)" "$(user $to)"
        return ($rewrite $from $to)
}       # end of crossbar
```

If you want to intercept specific mail messages, this function and the associated code in the crossbar and process functions will let you do it. There are three possible actions:

```
drop      - completely ignore this address
error     - return the specified error message
file      - append the message file to the specified file
```

Both the file and error actions require an argument, which necessitates the use of multiple-value return (i.e., return a list) in all cases.

If you don’t want to intercept anything, this function should return failure. The stub defined here is the usual case, you can override it in the host- specific cf file.

```
intercept (address) {
#       case "$(smtp_useratdomain "$address")" in
#       *@pdq*)         return (file /var/scr/pdq) ;;
#       rayan@csri.*)   return (drop) ;;
#       bitftp*@*)      return (error bounce) ;;
#       esac

        return 1
}
```

On mail from one local user to another, we don’t want to see all the long domain name extensions. This can cause problems with silly UAs, if it does you can just redefine {\tt intramachine} to call {\tt null} in your site or host-specific configuration files.

```
intramachine (address) {                # strip hostname if it came from here
        sift "$address" in
        (.*)@($hostname|$mydomain)
                address="$(condquote "\1")" ;;
        tfis
        return "$address"
}       # end of intramachine


null (address) {
        return "$address"               # surprise!
}
```

This is usually the default message-header address rewriting function. It is responsible for hostname hiding and qualification.

```
internet (address) {
        address="$(canonicalize "$address")"    # Canonicalize does local
                                                # hostname hiding...
        sift "$address" in
        (.*)<@(.+)>(.*)
                        #if [ $(deliver \2) ]; then     # hostname hiding
                        #       address="\1@${mydomain}\3"
                        #       break
                        #fi
                        address="\1@\2\3" # No hostname hiding...
                        ;;
        (.*)<(.+)>(.*)  address="\1\2\3" ;;             # defocus
        [^@]+
                # This is a local part address w/o any domains!
                address="$(condquote "$address")"
                address="$address@$mydomain"    # add our hostname
                ;;
        tfis
        return "$address"
}       # end of internet
```

A.3. `Process.cf`

This is the protocol switch function. It keys off the form of the filename to determine how to process a particular class of messages. It is expected that an internal function will be called to orchestrate the processing of the message and enforce proper semantics.

The file argument is the name of a file in the {\tt \$POSTOFFICE/router} directory.

```
process (file) {

        db flush pwuid
        db flush pwnam
        db flush fullname
        db flush hostexpansions
        db flush recipients
```

Since we cannot detect that the password database has been updated under our feet, we flush the cached information every once in a while (in this case, before every message).

```
        LOGMSG=''
```

The LOGMSG variable is used by the intercept facility (in {\tt crossbar.cf}) to make sure only a single copy of a message is saved when required. Each sender - recipient address pair can cause an intercept which can specify a file to save the message to. This variable is appended to elsewhere, and processed at the end of this function.

```
        case "$file" in
#       [0-9]*.x400)    x400 "$file" ;;
#       [0-9]*.uucp)    uucpfilter "$file" > /tmp/X.$$
#                       cat /tmp/X.$$ > "$file"
#                       rfc822 "$file" ;;
        [0-9]*)         rfc822 "$file" ;;
        core*)          /bin/mv "$file" ../$file.router.$$
                        return
                        ;;
        *)              /bin/mv "$file" ../postman/rtr."$file".$$
                        return
                        ;;
        esac

        [ $? ] && return 0      # Leave when they returned failure..
```

The file names in the {\tt \$POSTOFFICE/router} directory are determined by the parameter to the mail_open() C library routine. This case statement knows about the various message file types needed on your system, and arranges appropriate processing of each. The internal function {\tt rfc822} expects a file name as argument, and determines the semantics of the message and of the configuration code. For example, the {\tt router}, {\tt crossbar}, and {\tt header_defer} functions have semantics only because the {\tt rfc822} function knows about them. There are no other message formats supported in this distribution.

```
        log info: recipients $(db count recipients) $(elements $(cdr $(cdr $(cdr $(cdr $
'
```

For statistics gathering we print out the envelope information property list in its entirety, except for the file name, and the message id, both of which were logged earlier (in C code).

```
        for f in $LOGMSG
        do
                { echo "==${file}==$(rfc822date)==" ;
                  /bin/cat ../queue/"$file" } >> $f && log saved "$file" in $f
        done
}
```

This does the saving of intercepted messages into archive files.

A.4. Rrouter.cf

Most of the address routing processing is done here.

```
provide rrouter

. fqdnalias.cf # Pick that set of tools into here!

envelopeinfo=(message-id "<$USER.interactive@$hostname>" now 0)

: ${UNRESOLVABLEACTION:='error unresolvable'}

relation -bt selfmatch selfmatch

rrouter (address, origaddr, A, plustail, domain) {
        local tmp tee didhostexpand priv nattr a
        # local seenuucp seenbitnet
        # seenuucp=false
        # seenbitnet=false
        didhostexpand="";
# echo "rrouter: address=$address, origaddr=$origaddr" >> /dev/tty

        tmp=$(fqdn_neighbour "$origaddr" "$address" $A) &&
                return $tmp

        address="$(condquote "$address")"
```

We have troublesome addresses coming here...

```
        #       "|pipe-program"
        #       "|quoted string"@domain
        #       "foo > faa"@domain
        #       "fii < fuu"@domain
        #       "foo @ faa"@domain
        #       "|foo @ faa"
```

and we want to do correct focusing...

```
        ssift "$address" in
        # Now make canonical
        '"'(.*)'"'<(.*)
                address="\1\2"                  # defocus
                ;;
        '"'(.*)'"'>(.*)
                address="\1\2"                  # defocus
                break ;;
        ([\'"'].*[\'"'])<(.*)
                address="\1\2" ;;               # defocus
        ([\'"'].*[\'"'])>(.*)
                address="\1\2" ;;               # defocus
        # See that it does not start with a pipe ...
#       \|.+    # Looks like a pipe... Don't mutilate it!
#               break ;;
#       '"'[|].+        # Quoted pipe??   What the ...??
#               break ;;
        tfiss

        address=$(canonicalize "$address")

        ssift "$address" in
#       <in%>(.*)
#               return (((error vms-in-pros "in%\1" $A))) ;;
#       (.*)<@(.+)\.uucp>(.*)
#               seenuucp=true
#               address="\1$plustail<@\2>\3" ;;         # fix host.uucp!route
#       (.*)<@(.+)\.(bitnet|earn|netnorth)>(.*)
#               seenbitnet=true         # Strip off the (bitnet|netnorth|earn)
#               address="\1<@\2>\4" ;;          # fix host.bitnet!route
#        handle special cases.....
#       \\(.)   return (((local - "$address" $A))) ;;
        @       # handle <> form???
                tmp=(local user postmaster $A)
                return $(routeuser $tmp "")
                ;;
```

The following two are two approaches to the same problem, generally speaking we should use the SECOND one, but your mileage may vary... (Problems exist when WE are the target..)

```
#       (.*)<@\[(.)\]>(.*)
#               address="\1$plustail<@$(gethostbyaddr \2)>\3"
#               ;;
        (.*)<@\[(.)\]>(.*)
                # numeric internet spec
                if [ $(selfmatch "\2") ]; then
                        address="\1$plustail<@>\3"
                        domain="@[\2]"
                        plustail=""
                else
                        return (((smtp "[\2]" "\1$plustail@$(gethostbyaddr \2)\3" $A)))
                fi
                ;;
```

This is the end of the $1.2.3.4$ address case...

```
        (.*)<@(.*)\.>(.*)
                address="\1$plustail<@\2>\3"
                plustail=""
                ;;
```

Now massage the local info.

```
        (.*)<@(.*)($orgdomains)>(.*)
                address="\1$plustail<@\2$orgdomain>\4"
                domain="@\2$orgdomain"
                plustail=""
                ;;
        <@(.*)>[:,](.+)@(.+)
                if [ $(deliver "\1") ]; then # Source routed to our name?
                        return $(rrouter "\2$plustail@\3" "$origaddr" $A "" "")
                fi
                ;;
        <@($orgdomains)>[:,](.+)@(.+)
                return $(rrouter "\2$plustail@\3" "$origaddr" $A "" "")
                ;;      # strip organization
        (.+)<@(.+)>(.*)
                if [ $(deliver "\2") ]; then    # Do we handle this?
                        address="\1$plustail<@>\3"
                        domain="@\2"
                        plustail=""
                elif [ "\2" = "$hostname" ]; then # Is it at local host?
                        address="\1$plustail<@>\3" # (this is a backup test)
                        domain="@\2"
                        plustail=""
                fi ;;
        <@>.(.+)        # This plustail is propably wrong...
                return $(rrouter "\1$plustail" "$origaddr" $A "" "$domain") ;;  # try af
        (.+)<@>
                if [ -z "$domain" ]; then
                        domain="$mydomain"
                fi
                return $(rrouter "\1$plustail" "$origaddr" $A "" "$domain") ;;  # strip
        tfiss

#log "BITNET name=$bitnetname, address=$address"
        case $bitnetname in
        ?*)     tsift "$address" in
                (.*)<@(.*)\.(bitnet|netnorth|earn)>(.*)
                        address="\1<@\2>\4" ;;
                        # Strip off the (bitnet|netnorth|earn)
                tfist
                ;;
        esac
#log "BITNET name=$bitnetname, address=$address"
```

Resolve names to routes, get the actual channel name mostly from an external database.

```
        ssift "$address" in
        (.*)<@(.+)>(.*)
#log "neighbourg test: domain: \2, addr: $address"
                address="\1$plustail@\2\3"
                plustail=""
```

If you want to have the SMARTHOST to pick the routing for all non-local stuff, enable the following test case..

```
                # if [ "$SMARTHOST" ]; then
                #       return $(rrouter "$SMARTHOST!$(uucproute "$address")$plustail" "
                # else
                #       return ((($UNRESOLVABLEACTION "$address" $A)))
                # fi


                #if [ x$seenbitnet = xtrue ]; then
                #       address="\1@\2.bitnet"
                #fi

                didhostexpand=$(hostexpansions "\2")

                for method in $protocols
                do
                        tmp=$(${method}_neighbour "\2" "$address" $A) &&
                                return $tmp
                done

                #if [ x$seenuucp = xtrue ]; then
                #       if [ "$UUCPACTION" != "" ]; then
                #               return ((($UUCPACTION "\1@\2.uucp" $A)))
                #       fi
                #       tmp=$(routes_neighbour "\2.uucp" "$address" $A) &&
                #               return $tmp
                #fi

                #if [ x$seenbitnet = xtrue ]; then
                #       if [ "$BITNETACTION" != "" ]; then
                #               return ((($BITNETACTION "\1@\2.BITNET" $A)))
                #       fi
                #fi


                if [ "$SMARTHOST" ]; then
                        return $(rrouter "$SMARTHOST!$(uucproute "$address")" "$origaddr
                else
                        return ((($UNRESOLVABLEACTION "$address" $A)))
                fi
                ;;

        \\(.+)  # A back-quote prefixed userid (most likely)
                return $(rrouter "\1" "$origaddr" $A "$plustail" "")
                ;;

        /.+     # file
```

Well, it could be a slash-notated X.400 address too..

```
                return (((local "file.$origaddr" "$address" $A)))
                ;;
        \|.+    # pipe
                return (((local "pipe.$origaddr" "$address" $A)))
                ;;
        :include:.+ # ":include:" -alias
```

We must test this here, because the file-path after this prefix may have a dot.

```
                tmp=(local "$origaddr" "$address" $A)
                return $(routeuser $tmp "")
                ;;
```

Ok, from now on if we don’t have a domain set, we use {\tt \$mydomain}

```
        .*      if [ -z "$domain" ] ; then
                        domain="@$mydomain"
```

```
                fi
                ;;
        (.+\.[^+]+)(\+.+)  # Dotfull name with a plus!
                plustail="\2"
                address="\1"
```

Fall forward for the dotfull processing.

```
                ;;
        .+\..+  # A dotfull name
                tmp="$(fullnamemap "$address")" && \
                        return $(rrouter "$tmp" "$origaddr" $A "$plustail" "$domain")
                if [ $(newsgroup "$address") ]; then
                        return (((usenet - "$address" $A)))
                fi
```

Okay... Not in our special fullname/newsgroup-files, lets see if it is in the traditional one?

```
                if [ $(aliases "$address") ]; then
```

It can be found from the normal aliases, run the alias processing.

```
                        tmp=(local "$origaddr" "$address" $A)
                        return $(routeuser $tmp "$domain")
                fi
                return (((error norealname "$address" $A)))
                return (((error nonewsgroup "$address" $A)))
                ;;

        .*      # Now all the rest of the cases..
                tmp=(local "$origaddr" "$address$plustail" $A)
                return $(routeuser $tmp "$domain")
                ;;
        tfiss
}       # end of rrouter

routes_spec (domain, address, A) {
        local tmp channel rscshost

        sift "$domain" in
#       (bsmtp3nd|bsmtp3|bitnet2|bitnet2deliver2)!(.)!(.)
        (bsmtp3nd|bsmtp3|bsmtp3nd|bsmtp3rfc|bsmtp3ndrfc)!(.)!(.)
                return (((\1 "\2@\3" "$address" $A))) ;;
        (defrt1)!(.)
                channel=\1
                rscshost=\2

                tmp="$(uucproute "$address")"
                sift "$tmp" in
                .+!([^!]+)!([^!]+)
```

We are trying to gateway through a DEFRT1 domain(!)

```
                        #return (((error bitnetgw "$address" $A))) ;;
```

This will usually work anyway, sigh...

```
                        return (((bsmtp3 "mailer@$rscshost" "\2@\1" $A))) ;;
```

```
                ([^!]+)!([^!]+)
```

The destination domain is the next hop, so we're all happy.

```
                        return ((($channel "\2@$rscshost" "\2@\1" $A))) ;;
                tfis
                ;;
        ignore!.*
                break
                ;;
        smtp!
                ssift "$address" in
                (.*)@(.+)
                        return (((smtp "\2" "$address" $A)))
                        ;;
                tfiss
                ;;
        dns!
                ssift "$address" in
                (.*)@(.+)
                        return (((smtp "\2" "$address" $A)))
                        ;;
                tfiss
                ;;
        (.?)!
                return ((("\1" - "$address" $A)))
                ;;
        delay!(.)
```

NB! envelope info must also be defined in interactive mode.

```
                tmp="$(/bin/expr $(get envelopeinfo now) + "\1")"
                return (((hold "$tmp" "$address" $A))) ;;
        (.?)!([^!]+)
                return ((("\1" "\2" "$address" $A))) ;;
        (.?)!(.+)
                # BEWARE LOOPS
                return $(rrouter "\2!$(uucproute "$address")" "$address" $A "" "$domain"
                ;;
        tfis
        return 1
}

uucproute (address) {
```

This function turns any address into a pure-! form. It should not call any other functions, since random other functions call it. In particular it should not use rfc822route which itself uses uucproute.

```
        sift "$address" in
        (.*)<(.*)>(.*)          address=\1\2\3 ;;               # defocus
        (.+!)@(.+)              address=\1$(uucproute "@\2") ;;
        (.+)([,:]@)(.+)         address=\1!\3 ; continue ;;
        :include:[^!]+          return $address ;;
        @(.+:)([^:]+)           address=\1$(uucproute "\2") ;;
        (.+):(.+)               address=\1!\2 ; continue ;;
```

This won't work properly for e.g. utzoo!bar@gpu.utcs.toronto.edu because gpu.utcs also has an active uucp connection with utzoo. It will work properly in other cases though, so if we have to guess...

```
        #([^!])!(.+)@(.+)       if [ $(ldotsys \1) ]; then
        #                               address=\1!\3!\2
        #                       else
        #                               address=\3!\1!\2
        #                       fi ;;
        (.+)!([^!]+)%([^!%]+)@(.+)      # route!a%b@c -> route!c!a@b
                                address=\1!\4!\2@\3 ; continue ;;
        ([^@]+)@(.+)            address=\2!\1 ;;
        @(.+)                   address=\1 ;;
        (.+)!([^!]+)[%@](.+)    address=\1!\3!\2 ;;
        tfis
        return "$address"
}       # end of uucproute
```

Appendix B. Scheduler’s Configuration File Samples

Here are sample scheduler configuration files pulled straight out of the sources.

B.1. `scheduler.conf`

```
#
# Scheduler configuration file
#
# The scheduler reads this file on startup or when it receives a SIGUSR1 signal
#
# Every channel/host combination in recipient addresses will be sifted through
# the clauses matched in this file, picking up parameters until a clause that
# specifies a command.  Everything is free-form with three requirements:
# - Clauses (i.e. the channel/host pattern) start at the beginning of a line.
# - Clause contents (i.e. the parameters) start after some whitespace
# - Clause content keywords are matched case INSENSITIVE
# - Components are separated by whitespace.
#
# Within command=" ... " strings, following "variables" are known:
#  $host message’s host
#  $channel message’s channel
#  ${LOGDIR} ZENV variable LOGDIR (all ZENV variables supported)
#
# NB! For command paths, the $PATH is:  $MAILBIN/ta
# (Unless an absolute path is given for a command)
# for *running* things, the CWD is: $POSTOFFICE/transport/
#

#
# Note, there are three kinds of resource-pool limitation parameters
# which control when a given channel+host pair (thread) is NOT taken
# into processing:
#
#  MaxTA:  (Set in "*/*" clause)
#       GLOBAL parameter limiting the number of transport-agent processes
#       that the scheduler can have running at the same time.
#
# With this you can limit the number of TA processes running at the
# same time lower than maximum allowed by your OS setup.
#
# The scheduler detects the max number of FDs allowed for a process,
# and analyzing how many FDs each TA interface will need  -  plus
# reserving 10 FDs for the itself, result is “probed maxkids”.
#
#  MaxChannel: (default: “probed maxkids”)
#       Selector clause specific value limiting how many transport-agent
#       processes can be running on which the “channel” part is the same.
#       You may specify dis-similar values for these as well.  For example
#       you may use value ’50’ for all your ’smtp’ channel entries, except
#       that you want always to guarantee at least five more for your own
#       domain deliveries, and thus have:
#               smtp/*your.domain
#                       maxchannel=55
#       If the sum of all “maxchannel” values in different channels exceeds
#       that of “maxta”, then “maxta” value will limit the amount of work
#       done in extreme load situations.
#
#  MaxRing:  ( default: “probed maxkids” )
#       This limits the number of parallel transport agents within each
#       selector definition.    This defined the size of the POOL of
#       transport agent processes available for processing the threads
#       matching the selector clause.
#
#  MaxTHR:   ( max processes per thread; default: 1 )
#       This limits the number of parallel transport agents within each
#       thread; that is, using higher value than default “1” will allow
#       running more than one TA for the jobs at the thread.
#
# Do note that running more than one TA in parallel may also require
# lowering OVERFEED value.  (E.g. having a queue of 30 messages will
# not benefit from more TAs, unless they all get something to process.
# Having OVERFEED per default at 150 will essentially feed whole queue
# to one TA, others are not getting any.)
#
#  OverFeed:
# This tells how many job specifiers to feed to the TA when
# the TA process state is “STUFFING”  Because the scheduler
# is a bit sluggish to spin around to spot active TAs, it does
# make sense to feed more than one task to a TA, and then wait
# for the results.
#


#
# There are also a few flags directing various things
#
#  QueueOnly:
# Existence directs threads created under this clause *NEVER* to
# auto-start with any timeout mechanism.  Usage of ETRN methods
# is required for the thread to start!
#
#  WakeupRestartOnly:
# Existence directs threads created under this clause to start
# at the arrival of the first message to the thread, but in case
# the thread exists, and is in “WAKEUP DELAY” state, new arrival
# does not start the thread.
#
#  AgeOrder:
# This directs the queue *always* to run in strict arrival time
# order.  It might not make sense in all situations, though..
# Originally the system ran always by permuting the thread order
# before running TAs so that if some message causes the thread to
# hang, others will get processed past it.
#
#  ByHost:
# The command-line of the clause has “$host” expansion in it, OR
# the channel-part of the clause selector has wild-cards in it, (or
# the user decided to have “byhost” directive just for the fun of it.)
# Switching TAs in between threads might not be advised..
# (well, use of  "$host"  isn't advisable in general..)
# (also, having wild-cards at the channel-part is not advised..)
#
#  ByChannel:
# The command-line of the clause has  "$channel" expansion in it, (or
# the user decided to have "byhost" directive just for the fun of it.)
# This flag is not really used anywhere.
#

#
# ======== Some external parameters - name starts from column 0, and =====
# ======== always begins with "PARAM" ===================================

#
# MAILQv2 authentication database file reference:
# If you define this (like the default is), and the file exists,
# scheduler mailq interface goes to v2 mode.
# (Nonexistence of this file  A) leaves system running, B) uses MAILQv1
#  interface along with its security problems.)
#

PARAMauthfile = "${MAILSHARE}/scheduler.auth"


#PARAMmailqsock = "UNIX:/path/to/mailq.sock"
#PARAMmailqsock = "TCP:174"

# Time for accumulating diagnostic reports for a given message, before
# all said diagnostics are reported -- so that reports would carry more
# than one diagnostic in case of multi-recipient messages.
#PARAMglobal-report-interval = 15m

#
# ========================================================================
#

#
# Default parameter boilerplate, following values are in use in
# all operational channel/host clauses, unless overridden in them..
#
*/* interval=1m
 idlemax=4m # Max lifetime of idled TA before it is killed
 # expire messages after 3 days without full delivery
 expiry=3d
 # when the scheduler gets to the end of the retry sequence,
 # it starts over at some random point in the middle.  The
 # numbers are factors of the scheduling interval.
 retries="1 1 2 3 5 8 13 21 34"
 # no default limits on simultaneous transport agents or
 # connections to a particular host
 maxchannel=0
 maxring=20
 #
 maxta=0 # Let it be automagically determined
 #
 # default uid/gid of transport agents
 user=root
 group=daemon
 #
 # A flag telling about queue-order..
 #
 ageorder
 overfeed=150
 #
 # Possible nice/setpriority values in case one wants to run
 # the scheduler at higher scheduling priority, than TA programs:
 #
 # "priority" sets ABSOLUTE value, and requires setpriority(2)
 # system call.  "nice" is -- well: nice(2)
 #
 # nice=2
 ##priority=0
 #
 # "syspriority"/"sysnice" set the value for the scheduler process
 # itself, and are not inherited from the default boilerplate to
 # other parameter blocks.
 #
 # sysnice=-2
 # syspriority=-2

# Deferred delivery is handled by this transport agent.  Deferrals are low
# priority, but they tend to bunch up.  The 1 channel slot means there will
# be lots of contention, and typical checking intervals will be a bit higher
# than what is specified (due to waiting for a free slot).
hold/*
 interval=5m
 maxchannel=1
 command=hold

# BITNET delivery methods

defrt1/*
 maxchannel=3
 command="sm -c $channel defrt1"

bsmtp3/*
 maxchannel=3
 command="sm -c $channel bsmtp3"

bsmtp3nd/*
 maxchannel=3
 command="sm -c $channel bsmtp3nd"
bsmtp3rfc/*
 maxchannel=3
 command="sm -c $channel bsmtp3"

bsmtp3ndrfc/*
 maxchannel=3
 command="sm -c $channel bsmtp3nd"

#
# Local delivery: files, processes, user mail
#
```

```
# Parameterless "local/file*" will get same values, as
# "local/pipe*" immediately following it has !
#
local/file*
local/pipe*
 interval=5m
 idlemax=9m
 # Originally we had 3 hour expiry, but if your local system goes to
 # a fizz (freezes, that is), your local mail may start to bounce
 # before anybody notices anything...
 expiry=3d
 # want 20 channel slots, but only one HOST
 maxchannel=15
 maxring=5
 #
 # Do MIME text/plain; Quoted-Printable -> text/plain; 8BIT
 # conversion on flight!  (Can't use CYRUS, nor PROCMAIL here!)
 command="mailbox -8"

#
# This fallback "local/*" can be used to yield different local
# delivery mechanism -- mailbox / CMU cyrus IMAP server / procmail
#
# The latter two can not do deliveries to explicite files / pipes,
# thus you need the  "local/file*" and "local/pipe*" above.
#

local/*
 interval=5m
 idlemax=9m
 # Originally we had 3 hour expiry, but if your local system goes to
 # a fizz (freezes, that is), your local mail may start to bounce
 # before anybody notices anything...
 expiry=3d
 # want 20 channel slots, but only one HOST
 maxchannel=15
 maxring=5
 #
 # Do MIME text/plain; Quoted-Printable -> text/plain; 8BIT
 # conversion on flight!
 command="mailbox -8"
 # Or with CYRUS server the following might do:
 #command="sm -8c $channel cyrus"
 # Or with PROCMAIL as the local delivery agent:
 #command="sm -8c $channel procm"


# Sometimes we may want to PUNT all out to somewhere without regarding
# on what the routing said:
#
# smtp/*
# maxchannel=199
# maxring=5
# command="smtp -F [192.89.123.25] -l ${LOGDIR}/smtp.punt"

# This is a FAST EXPIRY test case.. Will always cause bounce, btw.
# (those machines are cisco routers, which don't have smtp-servers..)
```

```
smtp/*-gw.funet.fi
 maxchannel=0
 maxring=5
 expiry=1m
 interval=15s
 retries="1"
 skew=1
 command="smtp -s" # -l ${LOGDIR}/smtp"

smtp/*.rutgers.edu
 maxchannel=199
 maxring=10
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.edu
 maxchannel=199
 maxring=20
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.com
 maxchannel=199
 maxring=30
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.uk
 maxchannel=199
 maxring=8
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.ca
 maxchannel=199
 maxring=10
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.{se,dk,is,no}
 maxchannel=199
 maxring=20
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.de
 maxchannel=199
 maxring=10
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.gov
 maxchannel=199
 maxring=5
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.mil
 maxchannel=199
 maxring=5
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.net
 maxchannel=199
 maxring=10
 command="smtp -s" # -l ${LOGDIR}/smtp"
smtp/*.org
 maxchannel=199
 maxring=10
 command="smtp -s" # -l ${LOGDIR}/smtp"

# Within FUNET we have a bit longer expiry..
smtp/*funet.fi
 maxchannel=199
 maxring=9
 # maxta=2
 interval=10m
 retries="1 1 2 3 5 8 13 21 34"
 skew=1
 # Do FORCED MIME-decoding into C-T-E: 8BIT
 command="smtp -8sl ${LOGDIR}/smtp"

# Within our organization we care more about speed and capacity than connections
# The maxchannel value should be larger than the value used by smtp/*, to avoid
# some potential state and phase problems in the queues.
smtp/*.fi
 maxchannel=199
 maxring=20
 interval=10m
 retries="1 1 2 3 5 8 13 21 34"
 skew=1
 command="smtp -s" # -l ${LOGDIR}/smtp"

#
# Connections to the outside shouldn't duplicate effort so we only allow one
# per destination.
#
smtp/*
 maxchannel=199
 maxring=50
 command="smtp -s" # -l ${LOGDIR}/smtp"

#
# Special channel which LOGS all outgoing protocol sessions, so that
# admin can divert any domain to this channel in case logging is desired.
#
smtp-log/*
 maxchannel=199
 maxring=50
 command="smtp -c $channel -s -l ${LOGDIR}/smtp-log"

#
#  LMTP (RFC 2033) protocol driver with presumed "standard" port of 2525.
#

smtp-lmtp/*
 maxchannel=199
 maxring=20
 interval=1m
 retries="1 3 7 15"
 command="smtp -c $channel -M -x -p 2525 -s -l ${LOGDIR}/smtp-lmtp"

#
# These messages will go only into the queue, and need explicite
# SMTP mediated ETRN request, before they become flushed out.
#

smtp-etrn/*
 maxchannel=199
 maxring=20
 interval=1h
 retries="12"
 queueonly
 command="smtp -c $channel -s" # -l ${LOGDIR}/smtp-etrn"

#
# Destinations desired to use TLS (a.k.a. SSL) encryption can be
# run like this example shows.  There are two possibilities about
# REQUIRING the TLS encapsulation; that detail is configured inside
# the  “smtp-tls.conf” file.
#
smtp-tls/*
 maxchannel=199
 maxring=20
 interval=1h
 retries="12"
 queueonly
 command="smtp -c $channel -s -S ${MAILSHARE}/smtp-tls.conf" # -l ${LOGDIR}/smtp-tls"

# smtpx is a channel where the delivery is done without checking at MXes;
# rather only on A/AAAA (address) entries:
smtpx/*
 maxchannel=90
 maxring=10
 command="smtp -c $channel -x -s" # -l ${LOGDIR}/smtpx"

# Connections to places which sit behind broken firewalls, e.g. Cisco PIX
# versions with allowing EHLO to go thru with feature reply, but then
# rejecting all ESMTP protocol features listed at that reply...
smtp77/*
 maxchannel=199
 maxring=50
 command="smtp -c $channel -77 -s" # -l ${LOGDIR}/smtp77"

# Combination of smtp77 and smtpx
smtp77x/*
 maxchannel=199
 maxring=50
 command="smtp -c $channel -77 -x -s" # -l ${LOGDIR}/smtp77x"

# Connections to places we want to drive 8-bit-clean channel to
# independent of what EHLO tells (or does not tell)
smtp8/*
 maxchannel=199
 maxring=50
 command="smtp -c $channel -8 -s" # -l ${LOGDIR}/smtp8"

# Combination of smtp8 and smtpx
smtp8x/*
 maxchannel=199
 maxring=50
 command="smtp -c $channel -8 -x -s" # -l ${LOGDIR}/smtp8x"

# Error messages.  Delivery can be retried at leisure.
error/*
 interval=5m
 idlemax=2m
 maxchannel=5
 command=errormail

# UUCP delivery.  The "sm" transport agent picks the first host it sees and
# will select further recipient addresses with that host only.  We tell
# the scheduler this with the "byhost" boolean, to avoid a staggered delivery
# effect if the scheduler has to discover this on its own.
uucp/*  maxchannel=5
  command="sm -8c $channel uucp"

# News delivery.  Hostname is always the same here.
usenet/* maxchannel=2
  command="sm -8c $channel usenet"

# UBC EAN X.400 gateway.  See comment at UUCP.
ean/*  maxchannel=1
  command="sm -c $channel ean"

# BitBucket channel
bitbucket/*
  maxchannel=1
  command="sm -c $channel bitbucket"

smtpgw-*/*
  maxchannel=30
  maxring=30
  command="sm -8c $channel $channel"
```

B.2. scheduler.auth

```
#
# APOP-like authentication control file for the ZMailer scheduler.
#
# Fields are double-colon (’:’) separated, and are:
#   - Username
#   - PLAINTEXT PASSWORD (which must not have double-colon in it!)
#   - Enabled attributes (tokens, space separated)
#   - Addresses in brackets plus netmask widths:  [1.2.3.4]/32
#
# Same userid CAN appear multiple times, parsing will pick the first
# instance of it which has matching IP address set
#
# The default-account for ’mailq’ is ’nobody’ with password ’nobody’.
# Third field is at the moment a WORK IN PROGRESS!
#
# SECURITY NOTE:
#   OWNER:      root
#   PROTECTION: 0600
#
# Attribute tokens:
# ALL well, a wild-card enabling everything
# SNMP "SHOW SNMP"
# QQ "SHOW QUEUE SHORT"
# TT "SHOW QUEUE THREADS", "SHOW THREAD channel/host"
# ETRN "START THREAD channel host"
# KILL "KILL THREAD channel host", "KILL MSG spoolid"
#
# - "nobody" via loopback gets different treatment from
#   "nobody" from anywhere else.
#
nobody:nobody:SNMP QQ TT ETRN:  [127.0.0.0]/8 [ipv6.0::1]/128
nobody:nobody:SNMP ETRN:        [0.0.0.0]/0   [ipv6.0::0]/0
#watcher:zzzzz:SNMP QQ TT ETRN: [127.0.0.0]/8 [192.168.0.1]/32
#root:zzzzzzz:ALL:              [127.0.0.0]/8 [192.168.0.2]/32
```

Appendix C. Using ZMailer with Mailinglist Managers

```
%% \section{Using ZMailer with Mailinglist Managers}

Text to be inserted here.
```

Appendix D. Adding new transport agents

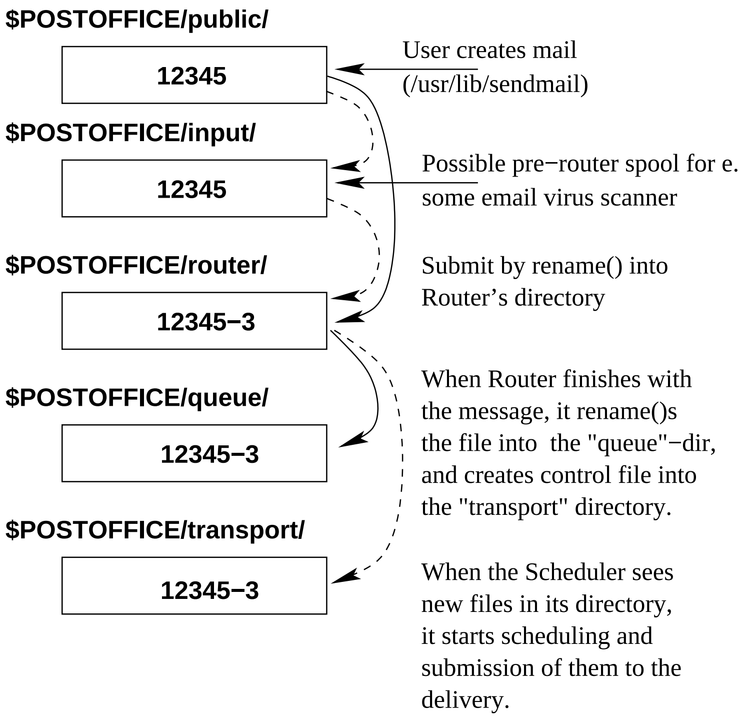
```
%% \section{Adding new transport agents}

Text to be inserted here.
```

Appendix E. Internal File Data Formats

E.1. ZMailer's Files and Formats

Figure E-1. ZMailer's `$POSTOFFICE/` directories and files



E.2. Envelope Header Lines

As the **router** picks up message files from a specific directory. Normally, message file names can be arbitrary valid file names, and indeed this is convenient when debugging. However, because the **router** daemon scans its own current directory, miscellaneous output from the *router* process may show up in this directory (e.g. profiling data, or core dumps (unthinkable as that is)). Furthermore, it is useful to be able to hide files from the **router** scanning (indeed the **router** may wish to do so itself).

When the **process** is scanning for message files, it only considers file names that have a certain format. Specifically, the message file name must start with a digit. This method was chosen to accomodate the message file names, as generated by the standard submission interface library routines, which will be strings of digits representing the message file's inode number.

A message file contains three sections: the message envelope, the message header, and the message body (in that order). he message body is separated from the previous sections by a blank line. The message body may be empty, and either of the message envelope or message header may be empty. The restriction on the latter situation, is that one of those sections must contain destination information for the message.

The message envelope and the message header have very similar syntax. The only difference is that while the message header must adhere to RFC822, the message envelope header fields are terminated by whitespace (“ ”) instead of a colon (“:”). The semantics of the two message file sections is quite different, and will be covered later.

The message envelope headers are used to carry meta-information about the message. The goal is to carry transport-envelope information separate from message (RFC-822) headers, and body. At first the message starts with a set of envelope headers (*-prefix denotes optional):

```
 *external \n
 *rcvdfrom %s (%s) \n
 *bodytype %s \n
 *errormsg \n
 *with %s \n
 *identinfo %s \n

Either:
  from <%s> \n
Or:
  channel error \n

 *envid %s \n
 *notaryret %s \n

Then for each recipient pairs of:
 *todsn [NOTIFY=...] [ORCPT=...] \n
  to <%s> \n

Just before the data starts, a magic entry:
  env-end \n
```

Then starts the message RFC-822 headers, and below it, the body.

The header fields recognized by ZMailer in the message envelope are:

`bodytype` *word*

not used. Compatibility with the sendmail feature

`channel` *word*

sets the channel corresponding to the message origin(*), usually as “`channel error`”

`comment` *string*

arbitrary comment

`env-end`

separator flag-word in between the envelope and the RFC822 headers

`env-eof`

alias to `env-end`

`errormsg`

Special internal flag-word telling that the message in question has been produced by **scheduler** or **errormail**, and is considered an error message.

This distinction can be used at routing to determine use of different default route lookup key for recipients in this case. See file `p-routes.cf`.

`envid` *xtext*

ESMTP DSN ENVID value

`external`

A flag-word indicating the external origin of a message

`from` *"address"*

a source address(*)

`fullname` *"phrase"*

sets the full name of the local sender

`identinfo` *"string"*

The SMTP server's ident lookup result, this does not guarantee anything about the sender though.

`loginname` *"word"*

requests using this mail id for the local sender

`notaryret` *"word"*

ESMTP DSN RET=*word*, either "`FULL`", or "`HDRS`"

`rcvdfrom` *"domain" (opt comment)*

An optional envelope entry, which sets "`Received:`" header's "`from`" field value.

This should *only* be used on messages that are originated thru "trusted" mechanisms, and especially *not* be used when the message is originated by some John Doe in the system. (E.g. this is reserved for **smtpserver** and friends, not for arbitrary users.)

`to` *"address-list"*

Normal recipient address list; usually used in form of listing one address in angle braces:

```
to <user@somewhere>
```

`todsn` *"phrase"*

ESMTP DSN recipient parameters. Note: this must be before the recipient "`to`" line for which this gives the extra parameters.

`user` *"local-part"*

Optional envelope entry telling who the message originating user was. The system is extremely suspicuous on this entry, and will check it against system account database, unless the spool file owner uid is known to belong to trusted users.

`verbose` *"filename"*

This optional envelope entry tells the router, what filename the sending client expects the subsystems to use as a feedback channel for reports concerning the file.

This "*filename*" is located into `$POSTOFFICE/public/` directory, and has been preopened by the same uid as has created the message spool file.

`via` *"word"*

An optional envelope entry that will define "`Received:`" header's optional "`via`" tag telling what physical transport mechanism was used.

Usually this entry is *not* used. (For an exceptions, see **rmail** and **listexpand** utility.)

`with` *"word"*

An optional envelope entry that will define "`Received:`" header's optional "`with`" tag telling what protocol was used.

Unlike RFC-822 tells, ZMailer supports only one "`with`" instance.

The (*)'s beside the descriptions indicate this is a privileged field. That is, the action will only happen if ZMailer trusts the owner of the message file (*Note Security: security. XREF ??). As with a normal RFC822 header, other fields are allowed (though they will be ignored), and case is not significant in the field name. The **router** will do appropriate checks for the fields that require it.

With this knowledge, we can now appreciate the minimal message file:

```
====================
to bond

====================
```

This will cause an empty message to be sent to *bond*. A slightly more sophisticated version is:

```
====================
from m
to bond
via courier
env-end
From: M
To: Bond
Subject: do get a receipt, 007!

You are working for the Government, remember?
====================
```

Notice that there is no delimiter between the message envelope and the message header. A more sophisticated example in the same vein:

```
====================
from ps/d-ops
to <007@sis.mod.uk>
env-end
From: M <d-ops@sis.mod.uk>
Sender: Moneypenny <ps/d-ops@sis.mod.uk>
To: James Bond <007@sis.mod.uk>
Subject: where are you???!
Classification: Top Secret
Priority: Flash

We have another madman on the loose.  Contact "Q" for usual routine.
```

```
====================
```

If the *Classification:* header is paid attention to in ZMailer, this requires that the **router** recognize it in the message header, and take appropriate action. In general the **router** can extract most of the information in the message header, and make use of it if the information is lacking in the envelope. The envelope headers in the above message are superfluous, since the same information is contained in the message header. Using the following envelope headers would be exactly equivalent to using the ones shown above (assuming the local host is *sis.mod.uk*):

```
====================
From Moneypenny <ps/d-ops@sis.mod.uk>
To James Bond <007@sis.mod.uk>
...
====================
```

ZMailer will extract the appropriate address information from whatever the field values are, as long as they obey the defined syntax (indicated in the list of recognized envelope fields above). ZMailer will complain in case of unexpected errors in the envelope headers.

The message body is not interpreted by ZMailer itself. As far as the **router** is concerned, it can be arbitrary data. However, certain *Transport Agents* may require limitations on the message body data. For example, the SMTP only deals with ASCII data with a small guaranteed line length.

E.3. Message Control File

A message control file is a file created by the **router** to contain all the information necessary for delivery of a message submitted in a corresponding message file. It is interpreted by the **scheduler**, which needs to know at all times which messages are pending to go where, and how. It is also interpreted by one or more *Transport Agents*, possibly concurrently, that extract the delivery information relevant to their purpose.

The concurrency aspect means that the *Transport Agents* must cooperate on a locking protocol to ensure that delivery to a particular destination is attempted by only one *Transport Agent* at a time, and a status protocol to ensure unique success or failure of delivery for each destination. There are potentially many ways to implement such protocols, but, in the spirit of simplicity, ZMailer uses a control file as a form of shared memory. Specific locations within each control file are reserved for flags that indicate a specific state for their associated destination address. The rest is taken care of by the I/O semantics when multiple processes update the same file.

Apart from necessary envelope and control information, a control file also contains the new message header for the message, which contains the header addresses as rewritten by the **router**. Since a message may have several destinations with incompatible address format requirements, there may be several corresponding groups of message headers. This will be illustrated by the sample control file shown in the following subsection.

E.3.1. Format

A control file consists of a sequence of fields. Each field starts at the beginning of a line (i.e. at byte 0 or after a Newline), and is identified by the appearance of a specific character in that location. This id character is normally followed by a byte containing a tag value (semaphore flag), followed by the field value.

Here is a simple control file produced by a test message, just before it was removed by the Scheduler:

```
====================
@ 0x00000007
i 24700
o 72
l <88Jan10.003129est.24700@bay.csri.toronto.edu>
e Rayan Zachariassen <rayan>
s local - rayan
r+          local - rayan 2003
m
Received: by bay.csri.toronto.edu id 24700; Sun, 10 Jan 88 00:31:29 EST
From:   Rayan Zachariassen <rayan>
To:     rayan, rayan@ephemeral
Subject: a test
Message-Id: <88Jan10.003129est.24700@bay.csri.toronto.edu>
Date:   Sun, 10 Jan 88 00:31:24 EST

s local - rayan@bay.csri.toronto.edu
r+          smtp ephemeral.ai.toronto.edu rayan@ephemeral.ai.toronto.edu 2003
m
Received: by bay.csri.toronto.edu id 24700; Sun, 10 Jan 88 00:31:29 EST
From:   Rayan Zachariassen <rayan@csri.toronto.edu>
To:     rayan@csri.toronto.edu, rayan@ephemeral.ai.toronto.edu
Subject: a test
Message-Id: <88Jan10.003129est.24700@bay.csri.toronto.edu>
Date:   Sun, 10 Jan 88 00:31:24 EST

====================
```

The id character values are defined in the `mail.h` system header file, which currently contains:

```
/* These are in order (roughly) what the router writes out. */
#define _CF_FORMAT      '@'     /* What format variant are we ?? */

#define _CF_FORMAT_TA_PID       0x00000001 /* At 'r' or 'X' lines */
#define _CF_FORMAT_DELAY1       0x00000002 /* At 'r' or 'X' lines */
#define _CF_FORMAT_MIMESTRUCT   0x00000004 /* The 'M' block       */
#define _CF_FORMAT_KNOWN_SET (_CF_FORMAT_DELAY1|_CF_FORMAT_TA_PID | \
                              _CF_FORMAT_MIMESTRUCT)

#define _CF_VERBOSE     'v'     /* log file name for verbose log (mail -v) */
#define _CF_MESSAGEID   'i'     /* inode number of file containing message */
#define _CF_BODYOFFSET  'o'     /* byte offset into message file of body */
#define _CF_LOGIDENT    'l'     /* identification string for log entries */
#define _CF_BODYFILE    'b'     /* alternate message file for new body */
#define _CF_ERRORADDR   'e'     /* return address for error messages */
#define _CF_OBSOLETES   'x'     /* message id of message obsoleted by this */
#define _CF_TURNME      'T'     /* trigger scheduler to attempt delivery now */
#define _CF_SENDER      's'     /* sender triple (channel, host, user) */
#define _CF_RECIPIENT   'r'     /* recipient n-tuple, n >= 3 */
#define _CF_DSNRETMODE  'R'     /* DSN message body return control */
#define _CF_XORECIPIENT 'X'     /* one of XOR set of recipient n-tuples */
```

```
#define _CF_RCPTNOTARY  'N'     /* DSN parameters for previous recipient */
#define _CF_DSNENVID    'n'     /* DSN 'MAIL FROM<> ENVID=XXXX' data */
#define _CF_MSGHEADERS  'm'     /* message header for preceeding recipients */
#define _CF_MIMESTRUCT  'M'     /* compacted MIME structure data for message */
#define _CF_DIAGNOSTIC  'd'     /* diagnostic message for ctlfile offset */

/* The following characters may appear in the second column after most _CF_* */
#define _CFTAG_NORMAL   ' '     /* what the router sets it to be */
#define _CFTAG_LOCK     '~'     /* that line is being processed, lock it */
#define _CFTAG_OK       '+'     /* positive outcome of processing */
#define _CFTAG_NOTOK    '-'     /* something went wrong */
#define _CFTAG_DEFER    _CFTAG_NORMAL   /* try again later */
```

There is one field per line, except for `_CF_MSGHEADERS`, and `_CF_MIMESTRUCT`, which have some special semantics described below.

The following describes the fields in detail:

`@` *hex-encoded-bitflagset*

This carries a hex-encoded bitflag set which is used by the **scheduler**, and *Transport Agents* to detect if the **router** produces files with incompatible features to what the latter programs know.

This is used to ensure that there stays a capability relation of:

```
router <= scheduler <= transport-agents
```

For example:

```
@ 0x00000007
```

`v` *relative-file-path*

Log file name for verbose log (**`mail -v`**):

```
v ../public/v_some_magic_tmpfile
```

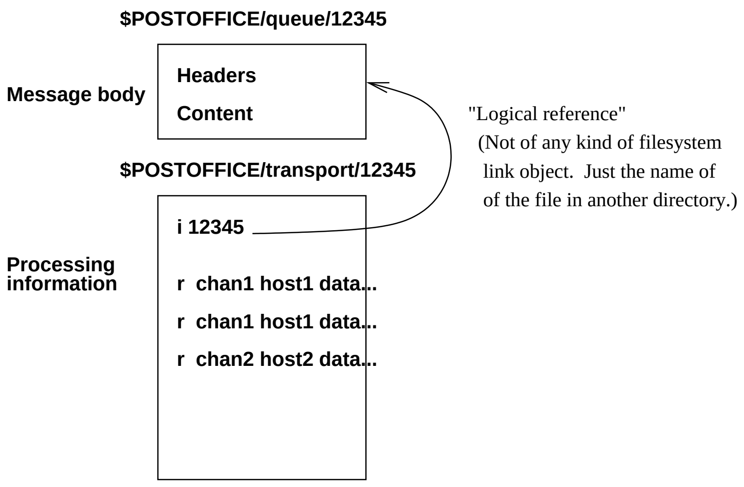
`i` *filename*

This field identifies the message file corresponding to this control file. It is the name of the message file in the *QUEUE* directory (`$POSTOFFICE/queue/`).

This is typically the same as the inode number for that file, but need not be. It is used by *Transport Agents* when copying the message body, and by the **scheduler** when unlinking the file after all of the destination addresses have been processed.

For example:

```
i 21456-789
```



`o` *decimal-number*

Specifies the byte offset of the message body in the message file. It is used by *Transport Agents* in order to copy the message body quickly, without parsing the message file.

For example:

```
o 466
```

`l` *message-id-string*

The field value is an uninterpreted string which should prefix all log messages and accounting records associated with this message. This value is typically the message id string.

For example:

```
l <88Jan6.103158gmt.24694@sis.mod.uk>
```

`b` *filename*

Alternate message file for new body. (*Currently not supported!*)

`e` *error-address*

Gives an address to which delivery errors should be sent. The address must be a RFC822 mailbox.

For example:

```
e "Operations Directorate" <d-ops@sis.mod.uk>
```

`x` *message-id-string*

Message id of message obsoleted by this.

`T`

This is mainly **smtpserver** created message directing the **scheduler** to trigger sending of given queue (or other parameter) right (resources permitting). This is mainly superceded by *MAILQv2* ETRN IPC mechanism.

For example:

```
T some.specific.domain (trigger originator IP address)
```

`s` *sender-quad*

This field specifies an originator (sender) address triple, in the sequence: previous channel, previous host, return address. It remains the current sender address until the next instance of this field.

Since there can only be one sender of a message, multiple instances of the field will correspond to different return address formats as produced by the `crossbar` algorithm in the **router**.

For example:

```
s smtp sis.mod.uk @lab.sis.mod.uk:q@deadly-sun.lab.sis.mod.uk
s uucp sisops lab.sis.mod.uk!deadly-sun.lab.sis.mod.uk!q
```

`r` *10-spaces rcpt-quad*

This field specifies a destination (recipient) address triple, in the sequence: next channel, next host, address for next host. Optional information to be passed to the *Transport Agent* may be placed after the mandatory fields; this currently refers to the delivery privilege of the destination address. Since the optional values of this field are only interpreted by the *Transport Agent*, changes in what the **router** writes must be coordinated with the code of the *Transport Agents* that might interpret this field.

For recipient processing interlocks, and delay report flags there is 6+4 spaces before the actual recipient address quad.

For example:

```
==123456ABCD....
r           local - bond 0
r           uucp uunet sisops!bond -2
```

`X`

One of XOR set of recipient 4-tuples. (*Not used so far.*)

`N`

DSN parameters for previous recipient.

For example:

```
==123456ABCD....
r           local - bond 0
N ORCPT=rfc822;bond NOTIFY=DELAY,FAILURE
```

`R`

DSN message body return control flag. (While this is stored once per every message

For example:

```
==123456ABCD....
r           local - bond 0
N ORCPT=rfc822;bond NOTIFY=DELAY,FAILURE
R HDRS
```

`n`

DSN `MAIL FROM<..> ENVID=XXXX` data.

For example:

```
==123456ABCD....
r           local - bond 0
```

```
N ORCPT=rfc822;bond NOTIFY=DELAY,FAILURE
R HDRS
n XXXX
```

`m`

Apart from a message body, a *Transport Agent* needs the message headers to construct the message it delivers. These message headers are stored as the value of this field.

Since message headers obviously can span lines, the syntax for this field is somewhat different than for the others. The field id is immediately followed by a newline, which is followed by a complete set of message headers. These are terminated (in the usual fashion) by an empty line, which also terminates this field.

In the following example, the last line of text is followed by an empty line, after which another field may start:

```
m
From: M
To: Bond
Subject: do get a receipt, 007!

s ...
r ...
```

`M`

This is another multi-line structure reserved for latter support of pre-scanned MIME structure data so that the transport-agents have easier work ahead of them when planning things like content transformations during the transport action. (12-Mar-2001)

`d`

This field is *not* written by the **router**. It is written by the **scheduler** or *Transport Agents* to remember errors associated with specific addresses. The field value has two parts, the first being the byte offset in the control file of the destination (recipient) address causing the error, and the rest of the line being an error message. The *Transport Agents* discover these errors and report them to the **scheduler**.

The **scheduler** will collect them and report them to the error return address (if any) after all the destinations have been processed.

For example: (FIXME! XREF to detail data ?)

```
d 878 No such local user: 'bond'.
```

It should be noted, that in sender and recipient fields the first two field values (channel and host) cannot contain embedded spaces, but the third field value (the address) may. Therefore, in the presence of extra fields, parsing within *Transport Agents* must be cautious and not assume that an address does not contain spaces.

As mentioned, the second byte of most fields are used for concurrency control and status indication. This tag byte can contain several values that indicate current or previous activity. The fields where this is relevant are the destination (recipient) address and diagnostic fields. The tag values are defined in the “mail.h” file mentioned previously, as follows:

```
#define _CFTAG_NORMAL ' ' /* what the router sets it to be */
#define _CFTAG_LOCK   '~' /* that line is being processed, lock it */
#define _CFTAG_OK     '+' /* positive outcome of processing */
```

```
#define _CFTAG_NOTOK  '-' /* something went wrong */
#define _CFTAG_DEFER  _CFTAG_NORMAL /* try again later */
```

The extract above is self-explanatory.

A message control file will normally contain a preamble that specifies information about the associated message file, the message body offset, an error return address, and a log entry tag. After this comes a repeated sequence of: sender address field, recipient address fields, and the message header corresponding to these recipients. After as many of these groups as are necessary, any diagnostic fields will be appended to the end of the control file. The restrictions on the sequence of addresses and message headers, are that a sender address field must precede any recipient address field, and a recipient address field must (immediately) precede any message header field, and no sender or recipient addresses may follow the last message header field.

E.4. Database File Formats

E.4.1. The `dbases.conf` file

Sample of `$MAILVAR/db/dbases.conf` file:

```
#|
#|  This configuration file is used to translate a semi-vague idea
#|  about what database sources (in what forms) are mapped together
#|  under which lookup names, and what format they are, etc..
#|
#|  This is used by  'zmailer newdb'  command to generate all databases
#|  described here, and to produce relevant  .zmsh  scripts for the
#|  router to use things.  The  'zmailer newdb' invocation does not mandate
#|  router restart in case the database definitions have not changed
#|  (reverse is true:  If definitions are added/modified/removed, the router
#|                     MUST be restarted)
#|

#|Fields:
#|     relation-name
#|         dbtype(,subtype)
#|              dbpriv control data (or "-")
#|                  newdb_compile_options (-a for aliases!)
#|                     dbfile (or "-")
#|                         dbflags (or "-") ... (until end of line)
#|
#| The  dbtype  can be "magic" '$DBTYPE', or any other valid database
#| type for the Router.  Somewhat magic treatment (newdb runs) are
#| done when the dbtype is any of: *DBTYPE/dbm/gdbm/ndbm/btree
#|
#| The "dbfile" need not be located underneath of $MAILVAR, as long as
#| it is in system local filesystem (for performance reasons.)  E.g.
#| one can place one of e.g. aliases files to some persons directory.
#|
#| At  dbflags  (until end of the line), characters ':' and '%' have special
#| meaning as their existence generates lookup routines which pass user's
#| optional parameters.  See documentation about 'dblookup'.
#|

#|Example:
#|
#|Security sensitive ones ("dbpriv" must be defined!)
#| aliases          $DBTYPE  0:0:644    -la $MAILVAR/db/aliases        -lm
#| aliases          $DBTYPE  root:0:644 -la $MAILVAR/db/aliases-2      -lm
#| fqdnaliases      $DBTYPE  root:0:644 -la $MAILVAR/db/fqdnaliases    -lm
#| userdb           $DBTYPE  root:0:644 -la $MAILVAR/db/userdb         -lm
#|
#|Security insensitive ones ("dbpriv" need not be defined!)
#| fqdnaliasesldap ldap   -  -   $MAILVAR/db/fqdnalias.ldap -lm -e 2000 -s 9000
#| fullnamemap      $DBTYPE  -    -l  $MAILVAR/db/fullnames    -lm
#| mboxmap          $DBTYPE  -    -l  $MAILSHARE/db/mboxmap    -lm
#| expired          $DBTYPE  -    -l  $MAILVAR/db/expiredaccts -lm
#| iproutesdb       $DBTYPE  -    -l  $MAILVAR/db/iproutes     -lmd longestmatch
#| routesdb         $DBTYPE  -    -l  $MAILVAR/db/routes       -lm%:d pathalias
#| thishost         $DBTYPE  -    -l  $MAILVAR/db/localnames   -lm%d  pathalias
#| thishost         unordered -   -   $MAILVAR/db/localnames   -ld    pathalias
#| thishost         bind,mxlocal - -  -                        -ld    pathalias
#| otherservers     unordered -   -   $MAILVAR/db/otherservers -lmd   pathalias
#| newsgroup        $DBTYPE  -    -l  $MAILVAR/db/active       -lm


aliases         $DBTYPE 0:0:644    -la $MAILVAR/db/aliases      -lm
fqdnaliases     $DBTYPE root:0:644 -la $MAILVAR/db/fqdnaliases  -lm%
userdb          $DBTYPE root:0:644 -la $MAILVAR/db/userdb       -lm

routesdb        $DBTYPE -   -l  $MAILVAR/db/routes      -lm%:d pathalias
thishost        $DBTYPE -   -l  $MAILVAR/db/localnames  -lm%d  pathalias

#| ================================================================
#|    Set of boilerplate tail-keepers, these lookups fail ALWAYS.
#| These are given because if user ever removes any of the relations
#| mentioned above, the generated "RELATIONNAME.zmsh" script won't
#| just magically disappear!
#| ================================================================

aliases         NONE - - - -
expired         NONE - - - -
fqdnaliasesldap NONE - - - -
fqdnaliases     NONE - - - -
fullnamemap     NONE - - - -
iproutesdb      NONE - - - -
newsgroup       NONE - - - -
otherservers    NONE - - - -
routesdb        NONE - - - -
thishost        NONE - - - -
userdb          NONE - - - -

#| NOTE:  mboxmap  MUST NOT exist at all if its secondary-effects
#|        are to be avoided!
```

E.4.2. Aliases File

For relation: `aliases`

Syntax of this file is simple: blank lines, and comments with “#” character at column 1 are ignored, the key is non-white-space string of characters terminating on double-colon + whitespace (actually ’"quoted string":’ is also valid key!), rest of the line (and possible continuation lines) are data.

```
postmaster: root
postoffice: root
MAILER-DAEMON: root
mailer:     postmaster
postmast:   postmaster

proto:  postmaster
sync:   postmaster
sys:    postmaster
daemon: postmaster
bin:    postmaster
uucp:   postmaster
ingress: postmaster
audit:  postmaster

autoanswer: "|@MAILBIN@/autoanswer.pl"

nobody: /dev/null
no-one: /dev/null
"no body": /dev/null
junk-trap: /dev/null

#test-gw: "|/..."
#test.gw: "|/..."
```

Doing expansion lists in sendmail(8) style is not suggested, although we certainly can do it. There is a better mechanism in the ZMailer to handle simple feats like these that sendmail(8) systems do by placing the file containing recipient addresses into the directory `$MAILVAR/lists/`. This directory must have protection of 2775 or stricter, and the listfile must have protection of 664 or stricter for *-request/owner-*/*-owner auto-aliases to work. — but to sendmail style lists:

```
listname: "/usr/lib/sendmail -fowner-listname listname-dist"
owner-listname: root # Well, what would you suggest for a sample ?
listname-owner: owner-listname
listname-request: root
listname-dist: ":include:/dev/null"
```

E.4.3. FQDNAliases File

This is syntactically alike the `aliases` database (the double-colon + whitespace terminate the key), rest of the line (and possible continuation lines) are data, however it can have some interesting keys:

`local@domain:`

Matches given address, including possible incoming `local+tag@domain` versions.

`@domain:`

This matches all addresses with given domain.

The result data may contain "`%1`" which is filled with `user` part of the input address, example:

```
@domain:  %1@domain2
```

It can thus be used to map e.g. `user@domain1` to `user@domain2`.

The main difference with sendmail's virtuser method is that this is generic *alias* type mapper, e.g. it can result in multiple addresses going out, or programs be driven, or...

E.4.4. Routes File

For relation: `routesdb`

Sampling here the default boilerplate "`$MAILVAR/db/routes`" file:

```
# Routing Configuration File
#
# Entries in this file are checked first by router.cf.
# They have the form:
#    name  channel!next_destination
# A leading . on the name indicates that all subdomains match as well
#
# We have TWO different fallback lookup tags:
#    .:ERROR  for cases where ERROR MESSAGES we generated are being routed
#    .        for general case
#
# This dictomy is due to need to route everything by explicite tables,
# EXCEPT in case of errors when '.' maps to 'error!something'
# ("We know to whom we route, others get error report back.")
#
# To generate runtime BINARY database of this source, issue command:
#   $MAILBIN/newdb $MAILSHARE/db/routes
# or in this directory with usual configuration:
#   ../bin/newdb routes
#

#
#  Sample route statements (and channels):
#
#   .foo  error!cannedmsgfilename
#   #       Canned error message from $MAILSHARE/forms/cannedmsgfilename
#
#   .bar  smtpx!
#   #       Send all traffic destined to any subdomain under this
#   #       suffix via "smtpx" channel to that domain
#
#   .bar  smtp-etrn!
#   .bar  smtp-tls!
#   .bar  smtp-log!
#   .bar  smtp77!
#   .bar  smtp77x!
#   .bar  smtp8!
#   .bar  smtp8x!
#   #       Ditto
#
```

```
#    .bar  smtpgw-xyz!
#    #       Drives genericish gateway function kit
#
#    junkdom      bitbucket!
#    myself       local!
#    news.domain  usenet!
#    uunode.dom   uucp!uunode
#
#    # Usual ISP smart-host setup
#    .       smtpx!ISP.smtp.gw
#
#    # Not so usual - fallback to error, except for error messages
#    .:ERROR smtp!
#    .       error!notourcustomer
#
```

E.4.5. Localnames

For relation: `thishost`

FIXME! WRITEME!

E.4.6. Otherservers

For relation: `otherservers`

FIXME! WRITEME!

E.4.7. Iproutes

For relation: `iproutesdb`

FIXME! WRITEME!

E.4.8. Fullnames

For relation: `fullnamemap`

This used to be a `firstname.lastname` keyed mapping database yielding login-ids, but these days this is superceded by ability to have dots in alias keys.

Example:

```
  firstname.lastname   loginid
```

E.4.9. Userdb

For relation: `userdb`

FIXME! WRITEME!

E.4.10. Expiredaccts

For relation: `expired`

FIXME! WRITEME!

E.4.11. Active (newsgroups)

For relation: `newsgroup`

FIXME! WRITEME!

E.4.12. Aliases.ldap

For relation: `aliases`

FIXME! WRITEME!

E.4.13. Fqdnaliases.ldap

For relation: `fqdnaliasesldap`

FIXME! WRITEME!

E.4.14. Mailbox File

Err... Uh.. What can be said ? The standard UNIX mailbox ?

FIXME! Or was this supposed to be the MBOXMAP thing ?

E.5. Scheduler Statistics Log

The statistics log reports condenced performance oriented information in following format:

tifileisd$tdapannel/$host ddt2
8 9 2 8 7 0 6 4 9 0 usenet/- 2 0 5

tifileid$state$channel/$host
dt1dt2

8928766238ok usenet/-
107

8928766244ok local/gopher-admin
101

8928766244ok smtp/funet.fi
105

8928766559ok smtp/utu.fi
1021

Where the fields are:

`timestamp`

The original spoolfile *ctime* (creation time) stamp in decimal.

`fileid`

Spoolfile name after the router has processed it.

`dt1`

The time difference from spoolfile ctime to scheduler control file creation by the router.

`dt2`

The time difference from scheduler file *ctime* to the delivery that is logged on.

`state`

What happened? Values: ok, ok2, ok3, error, error2, expiry

`$channel/$host`

Where/how it was processed.

E.6. Syslogged Log Formats

At *syslog* facility the system logs also material, if it has so been configured.

Different subsystems do different logs, they are described below.

E.6.1. Smtpserver's Syslog Format

The *smtpserver* may log in multiple formats:

INFO: connection from ...
WARN: refusing connection from ...
INFO: accepted id ... into freeze..
INFO: TASPID accepted from...

where TASPID: A spool-id that is valid throughout message lifetime in the system, and should be long-term unique, even. (Per system.)

EMERG: smtpserver policy database problem...
ERR: MAILBIN unspecified in zmailer.conf

E.6.2. Router's Syslog Format

The **router** does `syslog()` in following format:

```
taspid: from=<addr>, rrelay=smtprelay, size=nnn, nrcpts=nnn, msgid=str, delay=xx, xdela
```

Where:

`taspid`

The TA-SPOOL-ID — A spool-id that is valid throughout message lifetime

`from=`

the envelope source address

`rrelay=`

the message "rcvdfrom" envelope header reports.

`size=`

Total message size in bytes (envelope+headers+body)

`nrcpts=`

Number of recipients for this message

`msgid=`

The "Message-ID:" header content

`delay=`

Delay from message arrival to the system to this logging moment `xdelay=`

Delay during processing — tells how much time was spent to process the message.

E.6.3. Transport Agent's Syslog Format

The transport agents log in following format:

```
taspid: to=<addr>, delay=dd, xdelay=xx, mailer=mm, relay=rr (wtt), stat=%s msg
```

Here the fields are:

`taspid`

The ta-spool-id — A spool-id that is valid throughout message lifetime

`to=`

Destination address in whatever form the transport agent uses.

`delay=`

Delay from message arrival to the system to this logging moment

`xdelay=`

Delay during this processing attempt — tells how much time *this* time was spent to process the message.

`mailer=`

Tells what "channel" was used.

`relay=`

Reports on which host the message is relayed thru ("wtthost"), and for SMTP, also (in parenthesis) what was the relay's IP address.

`stat=`

What status was achieved: ok*, delayed, failed, ... *CHECK!*

msg

Arbitrary text line from whatever system is out there.

Appendix F. S/SL Language

The information in this appendix is based on "*Specification of S/SL: Syntax/Semantic Language*" by J.R. Cordy and R.C. Holt, December 1979 (Revised March 1980). Copyright (C) 1979, 1980 by the University of Toronto.

This appendix describes the S/SL language which is used within several scanners of the router system; RFC-822 object token scanner, and *zmsh* script language scanner to name the most important ones.

S/SL is a programming language developed at the *Computer Systems Research Group, University of Toronto* as a tool for constructing compilers. It has been used to implement scanners, parsers, semantic analyzers, storage allocators and machine code generators. S/SL has been used to implement compilers for Euclid, PT Pascal and Speckle, a PL/1 subset.

F.1. S/SL Introduction

S/SL is a procedure-based variable-free programming language in which the program logic is stated using a small number of simple control constructs. It accesses data in terms of a set of operations organized into data-management modules called mechanisms. The interface to these mechanisms is defined in S/SL but their implementation is hidden from the S/SL program.

S/SL has one input stream and one output stream, each of which is strictly sequential. These streams are organized into "tokens" each of which is read and written as a unit. An auxiliary output stream for error diagnostics is also provided.

F.2. S/SL: Identifiers, Strings and Integers

An S/SL identifier may consist of any string of up to 50 letters, digits and underscores (_) beginning with a letter. Upper and lower case letters are considered identical in S/SL, hence "aa", "aA", "Aa" and "AA" all represent the same identifier. `INPUT`, `OUTPUT`, `ERROR`, `TYPE`, `MECHANISM`, `RULES`, `DO`, `OD`, `IF`, `FI`, `END` and their various lower case forms are keywords of S/SL and must not be used as identifiers in an S/SL program.

An S/SL string is any sequence of characters not including a quote surrounded by quotes (").

Integers may be signed or unsigned and must lie within a range defined by the implementation. For example, this range could be -32767 to 32767 on a 16 bit machine.

Identifiers, keywords, strings and integers must not cross line boundaries. Identifiers, keywords and integers must not contain embedded blanks.

F.3. S/SL: Comments

A comment consists of the character "%" (which is not in a string) and the characters to the right of it on a source line.

F.4. S/SL: Character Set

Since not all of the special characters used in S/SL are available on all machines, the following alternatives to special characters are allowed.

- ! for |
- DO for {
- OD for }
- IF for [
- FI for]

F.5. S/SL: Source Program Format

S/SL programs are free format; that is, the identifiers, keywords, strings, integers and special characters which make up an S/SL program may be separated by any number of blanks, tab characters, form feeds and source line bound- aries.

F.5.1. S/SL: Notation

The following sections define the syntax of S/SL. Throughout the following, *{item}* means zero or more of the item, and *[item]* means the item is optional. The abbreviation "*id*" is used for identifier.

F.5.2. S/SL: Programs

An S/SL program consists of a set of definitions followed by a set of rules.

A program is:

```
[inputDefinition]        [outputDefinition]
[inputOutputDefinition]
[errorDefinition]        {typeOrMechanismDefinition}
RULES                    {rule}
END
```

F.5.3. S/SL: Input and Output Definitions

An *inputDefinition* is:

```
INPUT  ":"        {tokenDefinition} ";"
```

An *outputDefinition* is:

```
OUTPUT ":"        {tokenDefinition} ";"
```

An *inputOutputDefinition* is:

```
INPUT OUTPUT ":"  {tokenDefinition} ";"
```

A *tokenDefinition* is:

```
[string] ["=" tokenValue]
```

The *inputDefinition* section defines the input tokens to the S/SL program. The *outputDefinition* section defines the output tokens of the program. The *inputOutputDefinition* section defines those tokens which are both input tokens and output tokens of the program. Tokens already defined in the *inputDefinition* or *outputDefinition* sections must not be redefined in the *inputOutputDefinition* section.

The optional string which may be given in a *tokenDefinition* is a synonym for the token identifier and can be used in place of the identifier anywhere in the S/SL program.

Each input and output token is assigned an integer value for use in the implementation of the S/SL program. This value may be optionally specified in each *tokenDefinition*. The *tokenValue* may be specified as an integer or as the value of any previously defined identifier or string. If omitted, the value assigned to the token is the value associated with the previous token in the class plus one. The default value associated with the first input token and the first output token is zero. The default value associated with the first input-output token is the maximum of the last token defined in the *inputDefinition* section and the last token defined in the *outputDefinition* section. In this way the default input-output token values are unique with respect to both input tokens and output tokens.

F.5.4. S/SL: Error Signals

An *errorDefinition* is:

```
ERROR ":"         {errorSignalDefinition} ";"
```

An *errorSignalDefinition* is:

```
id ["=" errorValue]
```

Each *errorSignalDefinition* defines an error signal which can be signalled by the S/SL program. An integer error code value is associated with each *errorId* for use in the implementation of the S/SL program. This value may be optionally specified in each *errorSignalDefinition*. The *errorValue* may be specified as an integer or as the value of any previously defined identifier or string. The default value associated with an error signal is the value associated with the previous error signal plus one. The default value for the first error signal is 10 (errors 0 to 9 are reserved for S/SL system use).

F.5.5. S/SL: Type and Mechanism Definitions

Type and mechanism definitions may be grouped and inter-mixed to reflect the association of types and the operation definitions which use them.

A *typeOrMechanismDefinition* is one of:

- typeDefinition
- mechanismDefinition

F.5.6. S/SL: Types.

A *typeDefinition* is:

```
TYPE id ":"   {valueDefinition} ";"
```

A *valueDefinition* is:

```
id ["=" value]
```

Each *typeDefinition* defines a type of values for use as the parameter or result type of a semantic operation or as the result type of a rule.

Each *valueDefinition* defines a *valueId* in the type. An integer value is associated with each *valueId* for use in the implementation of the S/SL program. This value may be optionally specified in each *valueDefinition*. The value may be specified as an integer or as the value of any previously defined identifier or string. The default value assigned to a value identifier is the value associated with the previous value identifier plus one. The default value associated with the first *valueDefinition* in a type is zero.

F.5.7. S/SL: Mechanisms.

A *mechanismDefinition* is:

```
MECHANISM id ":"     {operationDefinition} ";"
```

Each *mechanismDefinition* defines the set of semantic operations associated with a semantic mechanism. The *mechanismId* itself is unused in the S/SL program. However, operation identifiers associated with a mechanism are by convention expected to begin with the mechanism identifier.

An *operationDefinition* is one of:

1. id
2. id "(" typeId")"
3. id ">>" typeId
4. id "(" typeId ")" ">>" typeId

Each *operationDefinition* defines a semantic operation associated with the mechanism.

- Form 1 defines an update semantic operation, which causes an update to the semantic data structure.
- Form 2 defines a parameterized update operation, which uses the parameter value to update the semantic data structure. The *typeId* gives the type of the parameter and can be any previously defined type.
- Form 3 defines a choice semantic operation, which returns a result value obtained from the current state of the semantic mechanism, which is used as the selector in a semantic choice. The *typeId* gives the type of the result and can be any previously defined type.
- Form 4 defines a parameterized choice operation. The first *typeId* gives the parameter type, the second the result type. Each can be any previously defined type.

Choice operations (forms 3 and 4 above) may be invoked only as the selector in a semantic choice.

F.5.8. S/SL: Rules

A rule is one of:

1.
   ```
   id ":"          {action} ";"
   ```
2.
   ```
   id ">>" typeId ":" {action} ";"
   ```

The rules define the subroutines and functions of the S/SL program. Rules may call one another recursively. A rule need not be defined before it is used. Execution of the program begins with the first rule.

- Form 1 defines a procedure rule which can be invoked using a call action.
- Form 2 defines a choice rule which returns a result value of the specified type. The {\tt typeId} can be any previously defined type. Choice rules may only be invoked as the selector in a rule choice.

F.5.9. S/SL: Actions

An action is one of the following:

```
1. inputToken
2. "." outputToken
3. "#" errorId
4. "{" {action} "}"
```

```
5. ">"

6. "[" { "|" inputToken {","    inputToken} ":" {action} }
               [ "|" "*" ":" {action} ] "]"

7. "@" procedureRuleId

8. ">>"

9. "[" "@" choiceRuleId {"|" valueId {"," valueId} ":" {action} }
               ["|" "*" ":" {action} ] "]"

10. ">>" valueId

11. updateOpId

12. parameterizedUpdateOpId "(" valueId ")"

13. "[" choiceOpId {"|"     valueId {"," valueId} ":" {action} }
               ["|" "*" ":" {action} ] "]"

14. "["     parameterizedChoiceOpId "(" valueId ")"
               "|" valueId {"," valueId} ":" {action} }
               ["|" "*" ":" {action} ] "]"
```

- Form 1 is an input action. The next input token is to be accepted from the input stream. If it is not the one specified, a syntax error is flagged. The *inputToken* may be an *inputTokenId*, an *inputOutputTokenId*, an *inputTokenString*, an *inputOutputTokenString*, or a question mark ("?"). The question mark is a special token which matches any input token.
- Form 2 denotes emission of an output token to the output stream. The *outputToken* may be an *outputTokenId*, an *inputOutputTokenId*, an *outputTokenString* or an *inputOutputTokenString*.
- Form 3 denotes the emission of the specified error signal to the error stream.
- Form 4 is a cycle or loop. Execution of the actions inside the cycle is repeated until one of its cycle exits (form 5) or a return (forms 8 and 10) is encountered. A cycle exit causes execution to continue following the nearest enclosing cycle. The cycle exit action is not allowed outside of a cycle.
- Form 6 is an input token choice. The next token in the input stream is examined and execution continues with the first action in the alternative labelled with that input token. The matched input token is accepted from the input stream.

  Each *inputToken* label can be an *inputTokenId*, an *inputOutputTokenId*, an *inputTokenString* or an *inputOutputTokenString*. A label can not be repeated nor appear on more than one alternative.

  The alternative labelled with an {\tt *} is the otherwise alternative. If the next input token does not match any of the alternative labels of the choice, execution continues with the first action in the otherwise alternative. If the otherwise alternative is taken, the input token is not accepted from the input stream, but remains as the next input token. After execution of the last action in an alternative of the choice, execution continues following the choice.

  If the next input token does not match any of the alternative labels and no otherwise alternative is present, a syntax error is flagged. For parsers written in S/SL, the default error handling strategy is to repeat the choice after modifying the input stream such that the next input token matches the first alternative. For compiler phases other than parsers, continued execution is undefined (the implementation aborts).
- Form 7 is a call to a procedure rule. Execution continues at the first action in the specified rule. When execution of the called rule is completed, either by executing the last action in the rule or by encountering a return action (form 8), execution is resumed following the call.

- Form 8 is a return action. It causes a return from the procedure rule in which it appears. A procedure rule may return explicitly by executing a return action or implicitly by reaching the end of the rule. A procedure rule must not contain a valued return (form 10).
- Form 9 is a rule choice. The specified choice rule is called and returns a value by executing a valued return action (form 10). The returned value is used to make a choice similar to an input token choice (form 6 above).

  Execution continues with the first action of the alternative whose label matches the returned value. If none of the alternative labels matches the value, the otherwise alternative is taken. Following execution of the last action in the chosen alternative, execution continues following the choice.

  Each alternative label in a rule choice must be a value of the result type of the choice rule. A label can not be repeated nor appear on more than one alternative.
- Form 10 is a valued return action. The specified value is returned as the result of the choice rule in which the action appears. The value must be of the result type of the choice rule. A choice rule may return only by executing a valued return action. A choice rule must not return implicitly by reaching the end of the rule. It must not contain a non-valued return (form 8).
- Form 11 is the invocation of an update semantic operation. Similarly, form 12 is the invocation of a parameterized update operation. The parameter value, which must be of the operation's parameter type, is supplied to the invocation of the operation.
- Form 13 is a semantic choice. The specified choice semantic operation is invoked and the returned value used to make a choice similar to an input token choice (form 6 above). Execution continues with the first action of the alternative whose label matches the returned value. If none of the alternative labels matches the value, the "otherwise" alternative is taken. Following execution of the last action in the chosen alternative, execution continues following the choice. Similarly, form 14 is a parameterized semantic choice. The parameter value, which must be of the operation's parameter type, is provided to the invocation of the choice operation.

Each alternative label in a semantic choice must be a value of the result type of the choice operation. A label can not be repeated nor appear on more than one alternative.

If the returned value in a rule choice or semantic choice does not match any of the alternative labels and no otherwise alternative is present, continued execution is undefined (the implementation aborts).

F.6. The Syntax of S/SL

> A program is:

```
[inputDefinition]      [outputDefinition]
[inputOutputDefinition]
[errorDefinition]      {typeOrMechanismDefinition}
RULES                  {rule}
END
```

An *inputDefinition* is:

```
INPUT ":"               {tokenDefinition} ";"
```

An *outputDefinition* is:

```
OUTPUT ":"             {tokenDefinition} ";"
```

An *inputOutputDefinition* is:

```
INPUT OUTPUT ":"       {tokenDefinition} ";"
```

A *tokenDefinition* is:

```
id [string] ["=" tokenValue]
```

An *errorDefinition* is:

```
ERROR ":"              {errorSignalDefinition} ";"
```

An *errorSignalDefinition* is:

```
id ["=" errorValue]
```

A *typeOrMechanismDefinition* is one of:

1. typeDefinition
2. mechanismDefinition

A *typeDefinition* is:

```
TYPE id ":"           {valueDefinition} ";"
```

A *valueDefinition* is:

```
id ["=" value]
```

A *mechanismDefinition* is:

```
MECHANISM id ":" {operationDefinition} ";"
```

A rule is one of:

```
1. id ":" {action} ";"
2. id ">>" typeId  ":" {action} ";"
```

An action is one of the following:

```
1. inputToken
2. "." outputToken
3. "#" errorId
4. "{" {action} "}"
5. ">"
6. "["     {"|" inputToken {"," inputToken} ":" {action} }
                ["|" "*" ":" {action} ] "]"
7. "@" procedureRuleId
8. ">>"
9. "[" "@" choiceRuleId {"|" valueId {"," valueId} ":" {action} }
                ["|" "*" ":" {action} ] "]"
10. ">>" valueId
11. updateOpId
12. parameterizedUpdateOpId "(" valueId ")"
13. "[" choiceOpId {"|" valueId {"," valueId} ":" {action} }
                ["|" "*" ":" {action} ] "]"
14. "["     parameterizedChoiceOpId  "(" valueId ")"
                {"|" valueId {"," valueId} ":" {action} }
                ["|" "*" ":" {action} ] "]"
```

Appendix G. RFC821

This will contain juicy bits regarding RFC 821, the SMTP protocol.

G.1. RFC821: "MAIL FROM:"

WRITEME! FIXME!

G.2. RFC821: "RCPT TO:"

WRITEME! FIXME!

G.3. RFC821: "DATA"

WRITEME! FIXME!

Appendix H. RFC822

This will contain juicy bits regarding RFC 822, the message *visible* header specification.

H.1. RFC822: "From:"

WRITEME! FIXME!

H.2. RFC822: "To:"

WRITEME! FIXME!

H.3. RFC822: "Cc:"

WRITEME! FIXME!

H.4. RFC822: "Subject:"

WRITEME! FIXME!

H.5. RFC822: "Date:"

WRITEME! FIXME!

H.6. RFC822: "Sender:"

WRITEME! FIXME!

Index