ZMOG

The ZMailer Operations Guide
Beta Release
Version 2.99.21

Matti Aarnio
mea@nic.funet.fi

Finnish University and Research Network
FUNET

Distribution

1. Introduction

ZMailer† is a mailer subsystem for the UNIX‡ operating system.

A mailer is in charge of handling all mail messages that are created on a system (typically a single host), from their creation until final disposition locally or by transfer to another system.

As such, the mailer subsystem (the Message Transfer Agent, a.k.a. MTA) must interface to local mail reading and composing programs (User Agents, a.k.a. MUAs), 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 benefited from experiences with earlier mailers.

ZMailer provides a capable, robust, efficient subsystem to do the job, which will excel in demanding environments, but is simple enough to fit easily everywhere.

While the ZMailer does take more system resources, than sendmail, it controls that resource consumption a bit better; SMTP input, and '`sendmail`'-compability module are rather lightweight processes that just feed a job spool file into local filesystem for other system components to tackle with.

Message routing decissions are done with a fixed set of processes, and final delivery is achieved with rather fine-grain controlled scheduler system.

1.1 Motivation and Heritage

Many of my“I” being usually Rayan Zachariassen around late 1980es; Matti Aarnio came in latter.easons for trying to improve the state of the art in message handling systems are based on the fact that the capabilities of available software hasn't changed for a long time, whereas the demands being placed on the software have steadily risen.

† Evoking references to The Mailer, That Mailer, or even Le Mailer, depending on reader's origin.
‡ UNIX was a Trademark of AT&T Bell Laboratories, but since then resold several times. Who owns it today, is for the reader to find out – if they care.

“I” being usually Rayan Zachariassen around late 1980es; Matti Aarnio came in latter. r

A few years ago, people were still dreaming of a world with consistent standards for addressing electronic mail and moving it around. Even though the consistency is constantly improving, it now seems apparent that there will always be needs that conflict with the ideal situation. This is very obvious to sites that interact with two or more types of networks.

For those sites that are directly attached to just one network, any degree of transparency in communicating with sites on other networks, must be provided by the software on a mail gateway. Most such software was not designed to perform the task of gatewaying messages between networks with heterogenous addressing and message standards.

ZMailer was primarily a reaction to Sendmail's disadvantages (which I shall mention), but also to the bad (and good!) points of several other mailers.

The design of ZMailer was often guided by my view of the poor choices or provisions of other mailers, as opposed to things they had done well.

This allows the design to draw from experience without limiting its creativity and use of new solutions.

To clarify my opinions somewhat, a commentary on the various mailers I know of should prove helpful:

1.1.1 Heritage: sendmail

The best available system to accomplish address manipulation in a heterogenous environment has previously been Sendmail, which was carried along on its relative strength for address manipulation.

Unfortunately, SendmailWe must agree that Sendmail v.8 has improved a lot from previously abundant v.5. . .as many design flaws that lessen its usefulness even in typical environments of our time.

Sendmail's major contribution to the art was the use of production rules to manipulate addresses and to guide the operations performed on a message.

It also popularized, and in certain environments pioneered, other functionality that turned out to be quite useful; for example the use of system-wide aliases file, as well as personal forwarding

We must agree that Sendmail v.8 has improved a lot from previously abundant v.5. . . h file, a widely available SMTP implementation, external and consistently treated delivery programs, etc.

Sendmail, in the right hands, can be quite a flexible tool to translate between the different conventions of various networks. Unfortunately this is accomplished by programming in an unfamiliar production language containing many magic features.

The learning time for doing this is very long, the effort involved is that of learning a completely new language and environment. Moreover, Sendmail has all major components built into a single large program.

Both of these design decisions have been acknowledged as mistakes by the author of Sendmail.

One of the big problems with Sendmail is its style of running all of the processing from submission/reception of the message to its final delivery in one process.

It makes no problem when the number of emails in processing is small, but when there are a lot of messages flowing, there are a lot of memory consuming, and load generating processes active. (With Sendmail v.8 there are reasonable load limiting functions available though...)

Its major shortcoming in comparison to the MMDF mailer is its primitive database facility and lack of caching. (Again the sendmail v.8 has since then improved things a bit.)

1.1.2 Heritage: MMDF, PMDF

MMDF is a comprehensive mail environment, including its own mail composition program, and of course a mailer.

There are too many parts to it (as a friend would say, it is a system, not a subsystem), and the address manipulation is only sufficient for a relatively homogenous environment.

It does have reasonable database facilities and caching, as opposed to Sendmail, and the concept of Channels.

However, knowledge about address semantics is distributed in several programs instead of being centralized.

PMDF is a smaller version of MMDF with correspondingly reduced features and flexibility. (Commercial PMDF is different story...)

1.1.3 Heritage: Upas

Upas is a curious approach to the problem. It lets the user do half the work of message routing, in a manner similar to PMDF on VMS systems.

It is entirely concerned with the message envelope, and leaves all message header munging to auxiliary programs, if appropriate. In fairness one should note this mailer was developed in an environment where most message headers were scorned, thus making this a reasonable approach ("optimize the normal case").

The Eighth Edition Upas had no database capability at all, but it did exhibit one useful characteristic: the routing decisions were made by passing the recipient envelope address through a set of regular expressions. This production rule approach is similar to what Sendmail does, but uses a more familiar mechanism and environment.

1.1.4 Heritage: Smail 3.0

The final, and most recently developed, mailer worth mentioning here is Smail3.0.Around 1988, since then Smail3.1 has been released with many differences...

It is intended as a program capable of replacing Sendmail in many situations. To a large extent it succeeds as this, and there are some nice ideas involved as well.

Its two major drawbacks are that it is not as easy to adapt to local needs as Sendmail is (compiled instead of interpreted rules and algorithms), and retaining Sendmail's single-program design.

It addresses database and caching issues, and seems generally like a nicer design in many respects, a bit like PMDF's configuration options in a Sendmail package.

XXX: Smail 3.1 ??

Around 1988, since then Smail3.1 has been released with many differences... .

1.1.5 Result: ZMailer

Until the recent increase in the demand for inter-network mail gatewaying, Sendmail's flexibility had quite adequately served to implement a gateway function between selected networks.

With increased variety of the normal address syntax and mail capabilities of connected networks, and more complex kinds of routing decisions becoming necessary, the existing mailers have been showing their age and their limits.

Also recent nearly exponential growth of the Internet has put serious demands on system performance, and there is readily a market for mailers processing more than million messages a day!

ZMailer is intended to give the mail administrator a software tool that fits the times.

1.2 Goals

Apart from the generic goals of robustness and efficiency, the following is a list of the specific goals of ZMailer:

- Fully RFC822/RFC976 compatible syntax and semantics.
- At a minimum provide Sendmail functionality from point of view of users and the system administrator.
- Make routing decisions based on original sender and path of the message.
- Not have hardcoded address rewriting and routing algorithms.
- A better user interface for the mail administrator than Sendmail.
- Interact properly with Internet Nameservers.
- Easily extensible to make use of new sources of data.
- Efficient enough to handle a large message volume, and to not significantly degrade its own or system performance when many messages are queued.
- Schedule delivery based on destination channel, destination host, or a combination of both.
- Able to do a better job, than Sendmail in our high load environments.
- Be small enough to be usable even on a smallest of the workstations as local program – well, "small" is rather flexible, but Linux version takes circa 1.0 MB disk, and 0.5 MB RAM when running in minimum configuration (Linux-1.2). For Sendmail the same figures are circa 0.3/0.3 MB…

For a while it has been apparent that Sendmail's approach to its task is not well-suited from several perspectives.

In particular, having a single program embody several conceptually independent functions is recognized as poor design.

In practice, merging queueing and delivery in one program causes a bottleneck for all messages when a particular delivery mechanism is slow. (Actually this is not an easy issue on any system.)

1.3 Design Summary

ZMailer is a multi-process mailer, using two daemon processes to manipulate messages. One of these processes is a router (or a set of routers), and it makes all decisions about what should happen to a message. The other daemon is a message queue manager, used to schedule 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 Router will process messages one at a time, as it finds them in a directory where User Agents submit their outgoing messages. (There can be parallel Router processes speeding up the processing.)

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 reconfigurable, since the control flow and address manipulation is specified by familiar shell script statements.

When the Router is finished, 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 Router 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.

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-compatible delivery programs.

The Scheduler also manages status reports, taking appropriate action on delivery errors and when all delivery instructions for a message have been processed.

A separate utility allows querying the Scheduler for the state of its mail 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.

2. Overview

This chapter deals with the life of a message, and what will happen to a message in the course of being processed. The processing activity is divided into four major phases that will be dealt with here.

These phases are: injection of a message into the mailer subsystem, message routing, message transport queueing and scheduling, and actual delivery of a message.

The phases communicate through the filesystem, by moving files from one directory to another.

All directories taking part in this communication are clustered under the ‘`POSTOFFICE`’ directory (‘`/usr/spool/postoffice`’), which is intended to also hold other maintenance information or directories for use by the system Postmaster.

For this reason, we shall refer to files within this hierarchy using the tilde-abbreviation (e.g. ‘`~/file`’ referring to the Postmaster’s ‘`$HOME/file`’ which is normally ‘`/usr/spool/postoffice/file`’).

2.1 Message Submission

A mail message is submitted to the mailer subsystem by depositing a “message file” in a particular ‘`ROUTER`’ directory (‘`~/router`’). There is a Sendmail replacement program which submits messages this way.

The messages are picked up by a daemon process scanning this directory, and processing all message files it finds in it.

To avoid problems with the daemon processing an incomplete or inconsistent message file, the message files are created in a separate ‘`PUBLIC`’ directory (‘`~/public`’), and then renamed into the ‘`ROUTER`’ directory.

A message file has 3 parts to it, the first part contains the envelope information for the message (if any). It consists of all the lines from the start of the file to the start of part 2 (exclusive) which are in the format of RFC822 Message Header lines *except* that there is no colon after the header field name.

An RFC822 Message Header is easily converted to an envelope header line by simply deleting the colon after the field name. As with message headers, various field names have specific semantics, and these will be discussed in detail later.

The second part of a message file is its RFC822 Message Header. The third part is the message body. Either of the envelope portion (part 1) or the message header portion (part 2) may be null. They are separated from the message body by an empty line, according to RFC822.

The standard UNIX conventions for files are obeyed (i.e. lines are terminated by a Newline character (`LF`)). The message body is never examined by the mailer itself, although transport/delivery programs must of course filter the message body appropriately for the destination. In other words, the message body may contain arbitrary binary data. The only restrictions are that the envelope and message header must obey the RFC822 lexical/syntax rules.

Once the message file has been written into the '`ROUTER`' directory, its content will never change until the system removes it after successful delivery.

A subroutine interface exists, which should be used by application programs or User Agents to submit messages.

The subroutine interface is properly part of the system C library ('`/lib/libc.a`'), and will be documented as such.

It is quite possible to submit messages by using the standard utilities to copy or move a file into the '`ROUTER`' directory.

Indeed, some maintenance functions the Postmaster should perform, and automatic resubmission of deferred messages, are most easily accomplished in this manner.

Note that the format of a message file allows a user to simply create a file that obeys RFC822 conventions, in order to submit a message. It also allows simple resubmission of a message which includes the UNIX standard '`From `' envelope header line (found in `Mail` format mailbox files), as this syntax will indeed be interpreted to represent envelope information.

2.2 Router

The daemon mentioned above, the one that processes message files appearing in the '`ROUTER`' directory, is called the Router process. It is the responsibility of this process to decide what to do with the message file, and pass this information on to the next stage (message transport/delivery queueing and scheduling). It does this by creating a Control File attached to the message file, which contains all the necessary information for the next stages to accomplish their work without detailed reference to the message file (i.e. without having to reparse it). When the Router process is finished processing a message file, it will relink it into a '`QUEUE`' ('`~/queue`') directory (it is flat directory, but it should not be a problem), and deposit the control file in a '`TRANSPORT`' directory ('`~/transport`') for processing by the next stage.

The Router will parse the message file contents, determine the boundaries between the various parts of the file, extract addresses and other information from the RFC822 format fields, and manipulate this information to determine the proper action for each destination address. The lexical and syntax analysis is carried out as a basic function within this process, but the semantics are determined by the contents of a configuration file for the Router. This configuration file is required to properly initialize the Router when it starts up, and furthermore defines functions that analyse an address, determine how to route it (given context information), and that can rewrite a message header address appropriately based on various context information.

The configuration file looks like a Bourne Shell script at first glance. There are minor syntax changes from standard `sh`, but the aim is to be as close to the Bourne Shell language as is practical. The contents of the file are compiled into a parse tree, 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 string-matching, and 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.

The Router makes use of environmental information to augment the information that may be contained in the message file. For example, the owner of the message file is the local user who submitted the message (which fact is used to check believability of some of the message header information), the message file modification time is used for local submission time, and the message file name is part of the synthesized message identification. Other envelope information, apart from the standard sender and recipient addresses, may be specified to augment the behavior of the mailer. For example, the standard library routines used to submit messages, may include code to pass along information from the submitting user's environment variables.

If something goes wrong when the Router processes a message file, its action depends on the severity and type of error. If for example there is a protocol violation of some kind, the Router may generate a rejection message sent to the originator of the offending message. The Router supplies the addressee and specific diagnostic messages corresponding to the error, and uses one of the canned files in the '`FORMS`' directory ('`~/forms`') for *everything else*. In particular this means such headers as '`From:`', '`Subject:`' and '`Cc:`' lines, and a generic comment on the class of error, are all taken from a standard form. This easily allows certain kinds of errors to be brought to the attention of the Postmaster or other maintenance person (by judicious use of the Carbon Copy field), and indeed different errors may be directed to different people.

For more serious problems, the message file is filed away in yet another '`POSTMAN`' directory ('`~/postman`'). This directory is where the Router will put any files that need manual attention by the Postmaster. The Postmaster may take corrective action (usually editing the message file), and resubmit the message file by simply moving it (using `mv`) to the '`ROUTER`' directory.

If something went wrong that may correct itself at a later time (for example if a database access indicates a temporary failure), the message file will be relinked into a '`DEFERRED`' directory ('`~/deferred`'). At some later time, these deferred message files may be resubmitted by moving them back to the '`ROUTER`' directory. This may be accomplished by a simple cron job. As indicated, the only problems that would cause this would be the lack of a resource needed by the Router. This may include out-of-space conditions on the disk, a database access timing out or returning a server failure reply, etc.

When everything does work properly, once a control file has been created in the '`TRANSPORT`' directory, and the message file moved to the '`QUEUE`' directory, the job of the Router is done for that message and it continues scanning its '`ROUTER`' directory for more work.

2.3 Queue Manager and Scheduler

The process that picks up control files from the '`TRANSPORT`' directory, is called the Scheduler. It is a daemon that orchestrates the flow of messages out from the mail subsystem. To do this, it maintains an internal model of which messages need to go where, and how, and passes the relevant information to the transport/delivery programs that it starts up. Various parameters associated with each transport/delivery program are controlled by a configuration file for the Scheduler. This configuration file is much simpler than the one for the Router, indeed it is a simple table format. A set of messages is selected using a channel/host specification pattern, and associated with each pattern one must specify a startup interval, command, and some related information, that will be used to deliver a selected message to the appropriate addresses. Specifying startup intervals for programs is the function which gives the Scheduler its name.

[XX: to do: rendezvous with Scheduler from unrelated program, e.g. uucico]

2.4 Transport Agents

A Transport Agent is responsible for doing the actual transport/delivery of a given message to a selected set of addresses. The selection of addresses is determined by the Transport Agent itself, perhaps using information about the name of the delivery channel or next host, passed on the command line. The messages each Transport Agent is asked to examine are determined by the Scheduler. A very simple protocol is run on the standard input and standard output of the Transport Agent, with a supervisor program (the Scheduler) choosing which control files the Transport Agent should know about. In turn, the Transport Agent returns status information about each of the addresses it processes in each control file, so the Scheduler can update its internal model of the collection of queued messages. As well, the Transport Agent is in charge of enforcing locking of a destination address while it is being processed, and the subsequent status update (success, deferral, error) in the message control file. These operations are performed in-place (and synchronously) on the contents of a control file.

All the actions and decisions made by a Transport Agent are driven entirely based on the contents of the control file. There is enough information about the original message file, that most Transport Agents will not need to reparse it. Standard Transport Agents exist for local mail delivery, SMTP/TCP, an error processing function, and interfacing with standard Sendmail “mailer”s.

2.5 The System Environment

Simplicity is an important thread in the design and implementation of ZMailer. This is with the hope that a simple (not simple-minded) design will encourage flexibility, elegance, and efficiency in the end product. If the design is done right, the result should fit in naturally with the UNIX environment, and will have some desirable side-effects: code portability (to UNIX variants and to other operating systems), and a smaller conceptual load for the person(s) maintaining the mail subsystem. This section is about the external interfaces to ZMailer; how it depends on the underlying system, how it interacts with it, and how the maintainers (the System Administrator and the Postmaster) communicate and interact with ZMailer.

All ZMailer activity is (largely by convention) confined to two directory hierarchies. One is used to keep program binaries and various databases, the other is a work area that is used when ZMailer does its job. For various reasons, this latter hierarchy is set up to mimic the various sections of a real postoffice, and indeed this analogy will reappear in a few user interface situations. The program/database locations may be spread out arbitrarily on your system. Unless there are good local reasons not to collect these files in one place, the following few conventions should be kept in mind:

The program/database directory is kept in '`/usr/lib/mail`' (first choice), or '`/usr/lib/zmail`' (in case the first choice is already taken). The program binaries of the Router and Scheduler portions of ZMailer are kept here, along with all program configuration files, and utility scripts. The databases, including the system aliases database, are kept in a '`db`' subdirectory. The program binaries of all the Transport Agents are kept in a '`ta`' subdirectory. If you like longer names, use '`databases`' and '`transports`' respectively.

In a mail/file server environment, mail clients only need a view of the '`POSTOFFICE`' directory hierarchy, so that User Agents can submit messages for processing and perhaps for the mail queue querying program to be able to read control files. The programs, configuration files, and databases stored under '`/usr/lib`' are only used by the mail server machine.

[XX: did I miss anything? Is this logical? should config files and programs be separate?]

An upcoming section (see section 2.8 [The Postoffice], page 15) will deal with the '`POSTOFFICE`' hierarchy in some detail. To motivate the issues dealt with there, we will first deal with the mechanics of sending a message.

2.6 User Agent support

To ease the task of interfacing directly to the ZMailer MTA, a C library routines are provided with ZMailer. The most important of these routines are used when submitting a message. They implement the message file name collision avoidance protocol, outlined in the previous subsection. An independent routine is provided to encourage proper quoting of the full names of users, as used in RFC822 message headers. This is an attempt at removing any excuse for "poetic license" on the part of User Agents, MTAs, and other systems (e.g. USENET News), where violation of the RFC822 specification in this regard is a frequent irritation. See [User Agent support], page 67, for details.

As mentioned earlier, information may be passed to ZMailer using envelope header lines in the message file. This includes a method for overriding the full name of the originating user, as found in the GECOS field of the password file entry for the user. It also includes a method for requesting an alternate login name (or rather *local-part*, in RFC822 terminology), which is of course subject to approval by security mechanisms in ZMailer. In order to promote a standard way of specifying these optional values, the message submission interface routines will seed the file with the appropriate envelope information to be interpreted by ZMailer. The user interface consists of the environment variables **FULLNAME** and **PRETTYLOGIN**, which are accessed through the standard `getenv()` routine. These environment variables need only be set by a user for all mail submitted through these interface routines to make use of the features.

2.7 Compatibility

Because ZMailer will often not be the first mailer installed on a computer, utility programs are provided to ease the transition between the different mail subsystems. The programs allow the change of MTA to be largely transparent to the User Agents, and other programs that interacted with the previously installed MTA. This allows conversion of such programs to be deferred to a more convenient time. If a slight performance penalty is acceptable, conversion may not be necessary at all.

In a Sendmail environment, there is just one critical program that needs to be replaced, namely the Sendmail binary itself. The only program, that is not a User Agent, which executes Sendmail directly, is the Rmail program (usually '`/bin/rmail`') which is conventionally used to transfer mail using UUCP. To avoid certain limitations of the standard Rmail programs, and at the same time gain performance by interacting directly with ZMailer, a new version of Rmail comes with the ZMailer distribution.

Most of the options on Sendmail are supported – when sensible to do so. Among them are a way to do a Verbose mode, and do other things that are common on Sendmail oriented environments.

2.8 The Postoffice

All of the message manipulation activity of ZMailer is confined to a directory hierarchy conventionally placed under '`/usr/spool/postoffice`'. This name reflects the kinds of activity carried out by ZMailer under the postoffice directory. The subdirectories under '`~`' are:

'`~/deferred`'
: a parking area for message files that cannot be processed due to temporary absence of resources needed by the Router. Such a situation would typically be due to a nameserver failure, or in case of unexpected I/O errors. Such message files can be

resubmitted by simply relinking them (use `mv`) to the directory scanned by the Router. This might be done periodically by a `find` command. Since the time granularity of `find` is rather coarse, a utility called `resubmit` is included with ZMailer to carry out exactly this task.

'`~/forms`' contains canned error and warning messages used by the ZMailer programs. Each file contains a prototype message, the only missing information is a destination address, and perhaps specific information that will be supplied by whichever programs make use of the form. In particular, specifying carbon-copy headers in these forms, allows the postmaster to get copies of mail automatically sent to users. Different types of messages may be carbon-copied to different people, if there are specialized postmasters on the system.

'`~/postman`'
is where ZMailer puts messages files that should be examined by the postmaster. Usually the postmaster is expected to take some corrective action, and resubmit the message. The actual reason ZMailer took this action will be found in the Router logs. As always, messages files can be resubmitted simply by relinking them into the directory scanned by the Router.

'`~/public`'
is the publically writable directory used by the standard message submission routines to create a new message file. When a message file has been properly created, it is relinked into the directory scanned by the Router. Empty files in this directory are often caused by improper handling of interrupts in a User Agent.

'`~/queue`' is the final resting place of message files, after the Router has processed them. This is where Transport Agents finds the message files when necessary. These files are eventually unlinked by the Scheduler.

'`~/router`'
is the directory scanned by the Router for new message files. From here, the message file goes to one of the '`~/queue`' (nominally), '`~/deferred`', or '`~/postman`' directories.

'`~/transport`'
is the collection of pending control files. The files here are unlinked by the Scheduler when all destinations have been processed. The Scheduler scans this directory periodically to detect new files that are not yet in the internal processing, and scans them in.

In a server-client machine environment, only the server machine needs to have this directory hierarchy. All the clients just need a view of the postoffice, for benefit of the User Agents. The message file name collision avoidance protocol should work properly across any remote filesystem. With this setup, the various ZMailer processes should only be run on the server machine.

3. Router

The Router is the smart half of ZMailer. All the other parts of ZMailer essentially just carry out instructions, as determined by the Router. Therefore, the Router is by far the most complex part of ZMailer. It must understand, in great detail, the structure of messages. It has to contain logic to manipulate portions of this structure, and, since many sites have different requirements and are in different network and mail environments, the logic used must be easily customizable. At the same time it should be efficient, since it is a bottleneck for message processing, and should cater to various services expected by System Administrators, Postmasters, and the users of a machine.

This description of the Router will begin by explaining how a message is submitted by User Agents and why a standard submission interface is a good idea. The structure of message files, and how that structure is analysed and used, is treated next. Then follows exposure of the mechanisms used to manipulate this information, and especially the tools available for the person configuring ZMailer to customize its behaviour. A final review of the details of the control logic will explain the reasons for various embedded behaviours of the ZMailer Router.

3.1 Message Submission

In the parlance of mail and message systems, ZMailer is an MTA, a Message Transfer Agent. It exists to process mail, similar to the function performed by a post office. As in the real world analogy, an MTA does not participate in the process of composing messages and getting them into the system. To do so, would correspond to your neighborhood postman taking dictation of your letters, and taking them along when leaving your house. In reality of course, people use a variety of tools to compose letters (quill pens, word processors, etc.), and to send them off. This functionality is embodied in a front end to the MTA, called a User Agent (UA for short). The choice of UA is a very personal one, and is usually not critical to the basic process of composing a message, sending it off, and getting it delivered properly.

Even though there may be many UA's in use on a computer system, there is usually only one MTA. The exceptions to this rule usually have to do with limitations in an MTA's capabilities. For example, a computer that can transfer mail using both the X.400 protocols, and the Internet protocols, may need two different MTAs to cover both protocol suites. Oftentimes, one of the two MTAs is a primary mailer, and takes care of all decision-making. The other mailers would then be treated by the primary MTA as a means for delivering messages to particular destinations, and these secondary mailers would be configured to punt any non-trivial traffic to the primary MTA. Both in the case of a User Agent, and in the cases of alternate MTAs, there must be a way to inject messages into the mail subsystem.

When you want to mail a letter, what do you do? Well, you drop it in a mailbox somewhere. ZMailer accepts messages the same way: you drop a file, containing your message, into a special submission directory. Like a postman (although rather more frequently), ZMailer scans this directory to pick up the new messages, and processes them. What happens to a message from then on is interesting in its own right of course, but presently we shall focus on how the message gets from a user into the mail subsystem.

The simplest way to "drop a file into a directory" is of course to actually edit a file in that directory. However, a program scanning such a directory would not be able to tell when you had finished writing your message and stopped editing the file. To do so, would require cooperation between the scanning program and all the programs that could conceivably create a file a portion at a time. The next most obvious method is to edit the message file in another location, and then simply copy or link it into the special message submission directory. Indeed, this is almost exactly the mechanism used. Actually, the copying program happens to be among those programs that may construct a new file piece by piece (a disk block at a time). For large files there is a vulnerable window between when the copy starts and when it finishes. If the submitter is unlucky, the message may be processed by the scanning program (the ZMailer Router) before it is completely written out. In fact this is not a problem due to the implementation policy of relinking message files. The real problem is if a partial message is completely processed and delivered and removed, with a corrupt message body.

To avoid problems, the only acceptable method is to make the complete message available to the Router at once. This is done by linking the message file into the directory being scanned. Due to the semantics of hard links and the UNIX filesystem, doing this requires that the message file is created on the same filesystem as the submission directory. ZMailer provides a publically writable directory, specifically for the purpose of creating message files before they are relinked into the submission directory. In fact, the submission directory itself is publically writable, so users can do this relinking themselves (e.g. with the `mv` command).

There are some security concerns with this approach. Because both these directories are writable, it is conceivable that a malicious user can cause problems for other users, for example remove their message files, or read or alter them if the file permissions allow. The solution to the latter problem, is obviously to ensure that file permissions do not allow people other than the originating user to access a message file. There are two solutions to the former problem; one is to maintain ignorance of the contents of the various directories, the other, better, method is to use a feature introduced in 4.3BSD – setting the sticky bit on the directories. The semantics of the sticky bit on a directory is to only allow the owner of a file to unlink it from a generally writable directory. If this function is not available, read permissions to the submission directory can safely be removed if only the standard message submission routines are used by the User Agents. This leaves us having to find a way to secure the directory used for creating files. Read permission to it cannot be withdrawn if the aforementioned standard routines are used (they are described later). Other things can be done, but obviously “ignorance” is not a reliable way of enforcing security. Perhaps if you notice the analogy with the common usage of ‘`/tmp`’ for various intermediate files, you will not consider security any more of a problem in this case. My recommendation would be to treat these directories the same way you treat ‘`/tmp`’. That is, if the directory sticky-bit semantics are available, use that feature. If not, try trusting your users enough to not cover up the directories. You should note that the possible danger is confined to removing message files. There is no way for a user to forge the origin of a mail message, since all validation of message origin is based on the ownership of the message file. Trusted user id’s may of course supply a different origin address.

There is only one other significant problem with this approach, which is the potential for name clashes of files in each of the two directories. This problem can be only be solved if all User Agents cooperate in using the exact same collision avoidance or collision resolution technique. What is really needed, by all the various ZMailer components operating on a message, is a way to get and hold a lock on the message. There are various ways to achieve this, and a kernel-based locking mechanism may seem appropriate in certain situations. However, given the realization afforded by an overview of the structure of ZMailer, it becomes clear that each message goes through states corresponding to the current processing stage. Instead of storing the state in the message file (or some other location associated with a particular message file), the state is encoded in the current location of a message file. For example, if the message file is in the submission directory, it means it is waiting for the ZMailer Router to process it. With this solution in hand, there still remains the original matter of avoiding file name clashes.

The best way of avoiding name clashes is to generate names that cannot clash. The only obvious unique property of a file is its inode number. Therefore, if one uses the inode number in the file name itself, name clashes will be completely avoided. This is a truism if the files are created on the same filesystem, and of course breaks down if that assumption is invalid. There is just one minor problem: the inode number of a file is not known until the file is actually created.

To resolve that Catch-22 situation, the message file must be created under one name, and then immediately renamed to its guaranteed unique name, using its inode number. The first name for a message file can safely be chosen by the same method used to find names for temporary files in ‘`/tmp`’. Now all the problems are solved with only two important assumptions: all message files are created by this same mechanism, and all message files are created on the same filesystem.

ZMailer will work with files that have been manually created and moved around, although this should only be done routinely by mail system maintainers.

3.2 Message File Format

As mentioned, 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 Router process is scanning for message files then, it only considers at 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). The 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 header fields recognized by ZMailer in the message envelope are:

`channel` *word*
: sets the channel corresponding to the message origin†

`from` *address*
: a source address†

`fullname` *phrase*
: sets the full name of the local sender

`loginname` *local-part*
: requests using this mail id for the local sender

`rcvdfrom` *domain*
: sets the host the message was received from†

`to` *address-list*
: a destination address list

`user` *local-part*
: sets the user the message was received from†

`via` *word*
: for RFC822 Received: header to be generated

`with` *word*
: for RFC822 Received: header to be generated

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
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:

† This is a privileged field. That is, the action will only happen if ZMailer trusts the owner of the message file (see section 3.10 [Security], page 38).

```
--------------------
from ps/d-ops
to <007@sis.mod.uk>
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.

3.3 Header Scanning and Parsing

Message header and envelope is scanned according to the lexical rules of RFC822 (and RFC976), and parsed according to the grammar rules of RFC822. RFC976 compatibility requires that the '`!`' and '`%`' characters be treated as specials (just like '`.`' and '`@`'). This behavior is enabled at compile time (by defining the `RFC976` preprocessor symbol), and is indeed enabled by default.

XXX: The only divergence from RFC822/RFC976 syntax is that comments are not allowed in certain locations within addresses, and comments and quoted strings may not span line boundaries. Neither of these are design limitations, they will disappear before final release. All other RFC822 constructs are properly recognized and supported.

The mentioned RFC documents serve to describe ZMailer behavior with respect to lexical scanning, tokenization, and parsing. In summary, based on the class of each character, a token stream is synthesized for each header. Various headers have defined semantics (e.g. the '`To:`' header contains an address list), which drive the parse of the token stream for that header. The headers that have specific semantics to the Router are:

```
Field name       RFC822 Syntax description      Class
-------------------------------------------------------------------
channel                 word                    Envelope
fullname                phrase                  Envelope
loginname               addr-spec               Envelope
rcvdfrom                domain                  Envelope
user                    mailbox                 Envelope
via                     word                    Envelope
with                    word                    Envelope

bcc                     #address                Recipient
cc                      1#address               Recipient
date                    date-time
encrypted               1#2word
errors-to               1#address               Sender
from                    1#mailbox               Sender/Envelope
in-reply-to             *(phrase | msg-id)
keywords                1#phrase
message-id              msg-id
received                received
references              *(phrase | msg-id)
reply-to                1#address               Sender
return-path             route-addr              Sender
return-receipt-to       1#address               Sender
sender                  mailbox                 Sender
to                      1#address               Recipient/Envelope
```

All the fields mentioned above are parsed by the Router. Some (e.g. '`In-Reply-To:`') are just parsed and not interpreted. Since the Router will complain about format violations, this is a way of enlightening people about what a particular field is not supposed to contain.

The '`Date:`' and '`Received:`' fields are interesting in that it is rather unusual for an RFC822 mailer to parse these fields. Indeed, whether or not they are parsed depends on the definition of compile-time preprocessor symbols (`CANON_DATE` and `CANON_RECEIVED` respectively). If a '`date`' header is parsed successfully, it will be printed using proper RFC822 date-time syntax when the message is delivered. For this to be useful, the date string parse in the Router must be rather flexible to recognize the endless variety of formats that exist, and it is.

The intention with parsing '`Received:`' headers is to prepare for the possibility of using the information in the trace headers to aid the routing algorithm. For now, parsed trace headers are output in a canonical format that follows RFC822, similar to what happens with parsed '`date`' headers. As long as the information is not used, there is no common reason to enable this feature. [XX: It should perhaps be possible to select these features on a per-message basis. any thoughts on this?]

Some of the header field names are tagged with what kind of addresses that header field contains. This information is used when searching for destination addresses when there are none specified in the envelope, and to know which headers contain addresses that must be sent through the address manipulation mechanisms of the Router.

3.4 Router Activities

The ZMailer Router has three basic functions that it must carry out on each message:

- Determining how to deliver a message to its destinations given in the message envelope.
- Rewriting message header and envelope addresses to accommodate the standards imposed by the method of delivery and the destination.
- Ensuring only properly formatted and standard-conforming (RFC822) messages leave the local system.

For everything but the syntax and semantics of addresses, the last goal is achieved by mechanisms internal to the Router. This is a reasonable approach since a standard is not something that adapts to local conditions. However, when pursuing the first two goals, many sites have found it invaluable to be able to modify the behavior of the routing function, and of the address rewriting function, to take local idiosyncrasies into account. The importance of this ability is very apparent to sites in complicated environments. Since ZMailer was partially motivated by the inadequacies of other mailers in such an environment, much effort has gone into the design of the configurable parts of the Router behavior. The wired logic of the Router is treated in a later subsection (See section 3.11 [Router Control Flow], page 39). Presently, we shall examine how routing and address manipulation is carried out, and the Router facilities which support these activities.

3.5 Routing Model

For routing purposes, one wants to derive three pieces of information from an address: where to send the message, how to send the message, and what to tell the immediate destination of the message about it. This is the information needed to properly transmit or deliver a message to its next destination.

The mechanism used to transmit a message may be regarded as a conduit (pipe, channel, circuit, etc.) between the local MTA and a remote MTA. In ZMailer terminology, such a conduit is called a Channel. A Channel is just a tag associated with a destination address for the message, and is used by the Scheduler to manage delivery of the message. Thus, a Channel is a concept (i.e. not associated with any particular program), and may be serviced by one or more Transport Agents. As far as the Scheduler is concerned, it is an uninterpreted classification of the message. For example, if there are different physical links to a remote MTA, different Transport Agent programs may serve the same Channel.

The Channel, or rather the Transport Agents serving a Channel, may need to know which remote MTA to deliver the message to. This is most often a hostname of a neighbouring host on a common network. If the Channel can only have one destination host (for example the local delivery Channel), a destination is superfluous. By convention, the Router will translate null destinations into the symbol '`-`' in a message control file.

The remote MTA will need to know what to do with the message, in the form of some envelope information. In RFC822, this information is embodied in an address for further delivery with respect to the remote host.

The Router must determine this triple (*channel*, *next-host*, *next-address*) for every address in the envelope, including the (single) origin address to be able to verify origin. If not for security, then to make sure that a proper RFC822 address was specified for the sender, and that a bogus address form is not passed on. To do this, the Router will call a function that takes an address as its argument and returns a triple. This function may be completely specified in a configuration file read by the Router, and its task is termed address resolution or routing.

While the `router` function rewrites envelope address as appropriate, there must also be a way to rewrite message header addresses. In Sendmail, this was done based entirely on which "mailer" (similar to a ZMailer Channel) the message was sent through. To do more sophisticated rewriting was not possible due to a complete lack of other information. If one wished to do different manipulations depending on the final destination of a message for example, it was almost impossible to do so (no variables or control flow in Sendmail rulesets). It was also impossible to do address manipulation or validity checking based on the origin of the message, since no such information was available.

The ZMailer Router remedies these and other shortcomings in several ways: the configuration language has control flow and variables, and the decision of how to rewrite each address is carried out with access to all the needed sender and recipient information. The word "decision" is used on purpose to indicate that the choice of rewriting method is divorced from the actual message header address rewriting process. What happens is that for each recipient, the Router calls a function passing the triples derived from the sender and the recipient address as arguments. The return value from this `crossbar` function (so named because a crossbar switch is the closest image, that came to mind, of what it does) includes the name of a function that is to be used for rewriting the message header addresses. This returned function is then called separately with all the addresses in the message header, and the results will be incorporated in the message header for the destination corresponding to the recipient triple. At the same time, while the routing function does generic resolution of an address into its corresponding triple, the crossbar function may modify the sender and recipient triples if necessary, and so serves as a cleanup or filtering function for the routing information. The crossbar function can also be completely specified in a configuration file read by the Router.

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.

3.6 Configuration File Programming Language

Whenever the Router process starts up, 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 the mailer, and this is primarily achieved by defining the `router` and `crossbar` 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. Indeed, 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 `for`, `while`, and `repeat` statements, no pipes, or I/O redirection.
- Case statement labels have no following ')', i.e. use

  ```
  case foo in
  pattern         action ;;
  esac
  ```

  instead of

  ```
  case foo in
  pattern)        action ;;
  esac
  ```

- Case label patterns use V8 (Eighth Edition UNIX) regular expression syntax (`egrep`-like).
- 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-'' value from a function. The following are all legal `return` statements:

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

- Variables are dynamically scoped, the only local variables are the ones in a function's parameter list. Only the first value of a multiple-value return may be assigned to a variable. All values are strings, 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 end of a *word*. 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)))`.

- The forms `${variable:=value}`, `${variable:-value}`, and `${variable:+value}` are supported. The special form `${string:relation}` returns the value of `relation(string)`, implementing a database lookup function.
- Patterns (in case labels) are evaluated once, the first time they are encountered.
- 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.

- Various standard Bourne shell functions do not exist built in.

There are currently only two entry-points (i.e. magic names known to the Router code) in the configuration file, namely the `router` and the `crossbar` functions.

The `router` function is called with an address as argument, and returns a triple of (*channel*, *host*, *user*) 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 (null if local delivery), and the address the receiving agent should transmit to.

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-triple and a receiver-triple as arguments (six parameters all together). It returns the new values for each element of the two triples, 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 one more magic symbol the Router knows about, which is (optionally) defined by the configuration file. That is the name of the definition of the alias database, a protocol which will be dealt with in the subsection explaining the database lookup mechanism hinted at above.

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 label
- `break` exits a case statement

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

`import` defines a function name that refers to an external program

`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:

`getzenv` retrieves global ZMailer configuration values

`echo` emulates '`/bin/echo`'

`exit` aborts the Router with the specified status code

`hostname` internal function to get and set the system name

`trace` turns on selected debugging output

`untrace` turns off selected debugging output

`[` emulates a subset of '`/bin/test`' (a.k.a. '`/bin/[`') functionality

The `relation` function is described in a later section (See section 3.8 [Database Interface], page 31), and the `trace` and `untrace` functions are described in connection with debugging (See section 3.12 [Logging], page 43).

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.

3.7 Address Manipulation

Most of the flexibility of Sendmail derives from its production-rule model for address rewriting.

Very loosely, the concepts of rulesets in Sendmail correspond to the functions of the ZMailer Router configuration file programming language, and the rules themselves correspond to the case label bodies of the case statements in our language.

Addresses are represented as string values in this language, no different from any other strings. Therefore, addresses can be assigned as the value of a variable, or passed as an argument to a function. The way to do address rewriting is to modify the value of a variable chosen to contain the *current* address (in the sense of the Sendmail rewriting process). In keeping with the production rule model, much of the address rewriting is typically done within case statements, whose semantics have been tailored for this activity.

A `case` statement in the configuration file language has almost the same syntax as the Bourne shell case statement. However, its semantics are different, in that it is similar to the philosophy of (Sendmail) rulesets. That is, the normal action is for an address to "enter" at the top, and for each case label (rule) that matches, the case label body (action) is executed. This is carried on sequentially for each case label (rule-action pair) in the case statement (rule set), unless the normal continuation action is modified by a control statement.

As opposed to Sendmail, where a rule is retested until it fails, a case label pattern is only retested if the case label body calls the special function `again`. This change was made because it is frequently a waste of time to retest a pattern match when one has just modified the string to be matched against. Sendmail does provide a way to continue to the next rule-action pair, but since it is not the default behavior, it is often not used in many of the places it should be used. As a way of reducing the consequent waste of time, the default behavior has been changed.

The other special function that is specific to case statements, is `break`. It is used with the same semantics as if within a C language looping construct or switch statement, i.e. to exit the case statement and continue with the statement after it. Of course, a `return` statement will completely return from its enclosing function at any time.

Case conditions usually are not just a simple constant string; they will usually contain a variable expansion and perhaps a function call. The value of such a condition changes as the variable(s) it depends on changes. When doing repeated case label pattern matching with the condition string value, it would be rather unsavory to reevaluate the condition expression every time. If no antecedent variable has changed value, obviously the expression will not change its value either. To avoid this unnecessary effort, the case condition is only reevaluated when any variable it depends on has been assigned to, and then of course only when the current expression value is actually needed.

Some final, and very important points: Even though the case label patterns look like normal regular expressions that one can find in editors and other system utilities, the pattern matching in the Router is token-based, rather than character-based. The tokens are of course the RFC822 tokens scanned from the value of the condition expression. This is done to avoid surprises from simplistic patterns, and to cut down on unnecessary verbosity in describing an address when using the normal regular expression semantics. Another thing that helps the matter, is that all case label patterns are anchored at the beginning and end of the string. An anchored pattern easily simulates an unanchored pattern, but not vice versa. In patterns, parentheses are used to group a number of alternates, and are also used to bracket portions of the pattern, so the corresponding tokens in the matched string can later be referred to. To avoid introducing another special character (backslash, conventionally used to refer to selected portions of the matched string), the semantics of the $-prefix notation are extended to handle this need. If a '$' is followed by a digit N, this is expanded as the value of the portion of the matched string selected by the N'th group of parentheses in the pattern.

To give an idea of how a case statement looks, here is a code fragment:

```
case $hostname in
.+\.(edu|gov|mil|oth|org|net|ca|dk|uk)      # add toplevels as you please
        break ;;                            # do nothing
.*      hostname = $hostname.$orgdomain ;;  # default domain
esac
```

3.8 Database Interface

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 Net 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 communicate to 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 string printed by the relation function when called the wrong way:

```
Usage: relation -t dbtype [-f file -s# -b|n -l/u -d driver] name
```

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:

`hostsfile`
: '`/etc/hosts`' lookup using `gethostbyname()`.

`unordered`
: the database is a text file with key-datum pairs on each line, keys are looked up using a sequential search.

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

`dbm`
: the database is in DBM format (strongly discouraged).

`ndbm`
: the database is in NDBM (new DBM) format.

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

A subtype is specified by appending it to the database type name separated by a slash. For example, specifying `bind/mx` as the argument to the '`t`' option will store away '`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 some types of the databases there is no practical access to any particular database file, these include `hostfile`, and `bind`. The use of the `dbm` format is strongly discouraged, since a portable program can only have a single DBM database associated with it. For YP (nee, "NIS") database the optional file defines NIS-domain.

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 '`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. Currently the only legal argument to this option is `pathalias`, specifying a driver that searches for the key using domain name lookup rules.

The final argument 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.

Some sample relation definitions follow:

```
if [ -f /etc/named.boot ]; then
  # relation -nt bind,cname -s 100 canon  # T_CNAME canonicalize hostname
    relation -nt bind,any -s 100 canon    # T_ANY 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
```

The above fragment defines a set of relations that can be accessed in the same way, using the same names, independent of their actual definition.

```
# We maintain an aliases database in the following format. Note: the
# 'aliases' db name is magic to the internal alias expansion routines.
if [ -f $MAILBIN/db/aliases.dat ]; then
    relation -t ndbm -f $MAILBIN/db/aliases aliases
else
    relation -t ordered -f $MAILBIN/db/aliases.idx aliases
fi
```

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. 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 the 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.

Finally, some miscellaneous definitions that illustrate various possibilities:

```
relation -t unordered -f /usr/lib/news/active -b newsgroup
relation -lmt ndbm -f $MAILVAR/db/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
```

Here, 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).

3.8.1 Using a Pathalias Database

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. It generates key-value pairs of the forms:

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

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:

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

# pathalias database lookup function
padblookup (name, path) {            # path is a local variable
        path = ${$name:padb}
        case "$path" in
        ((.+)!)?([^!]+)!%s
                if [ $3 == $name ]; then
                        path = $2!$3
                else
                        path = $2!$3!$name
                fi
                ;;
        .*%s.*  echo illegal route in pathalias db: $path
                ;;
        esac
        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`' flag given in the relation definition.

3.9 Mail Forwarding

Although more interesting and useful models exist, the mail forwarding functionality of ZMailer has been designed to generally emulate the interface and behaviour of Sendmail. The mechanisms that accomplish this are likely to be generalized in a future version.

If a relation named `aliases` is defined by the configuration file, then the data returned by a lookup in that database is assumed to be a printed decimal representation of the byte offset of the definition of the alias in a separate file. In other words, the `aliases` relation associates a particular local-part, with an index into another file that contains the actual alias definition. The name of this other data file is constructed from the name of the file associated with the `aliases` relation, typically it will be '`aliases.dat`'.

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 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.

There exists a special mechanism to do address expanding aliasing on a mailing-lists, however it doesn't really take into account all needs that mailinglist might have – like reader rewrite, moderation, etc. non-trivialities. If you need more, than simple alias-like address expansions, do look at various Mailing-List-Manager softwares.

The mechanism to do list-aliases is to look up for a file in directory '`$MAILVAR/lists/`'. If there exists a file with looked up name, then the address expansion is done by reading that file for recipient addresses in RFC822 format (just like with '`:include:`'d file). Furthermore, all recipients produced from that address are sent with '`listname-owner`' as the sender, and error catcher…

Related to the list-alias mechanism is test for `*-owner`, and `*-request` local-parts; if `basename somelist-owner -owner` resolves to a file in the '`$MAILVAR/lists/`'-directory, the owner of the file is taken to do handle the list.

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.

3.10 Security

Having local-parts that allow delivery to arbitary files, or 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 “trustworthyness” of that address is necessary so 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 trustworthyness chosen by [XX: the present incarnation of] ZMailer, is simply a user id (uid) value representing the source of the address.

When a message comes in from a non-local host, 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 loses anyway.
- There is a window of vulnerability between when the permissions are checked, and the delivery is actually made. This is the best argument I have heard so far for embedding the local delivery program (currently a separate Transport Agent) in the Router.

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 origin of a message without access to a trusted account.

3.11 Router Control Flow

The following few pages use pseudo-code to describe the algorithm that produces a control file (containing delivery instructions and the new message headers) from a message file. This algorithm is implemented in a C function called `sequencer()`, an apt description of how it orchestrates the various parts of the ZMailer Router to implement the semantics of RFC822 message processing.

```
sequencer(message file name)
{
    Parse envelope and message header from the message file

    if (hostname has been set)
        Stamp the message with a trace header

    Determine if message contains Resent-* headers or not
    (from here on, only pay attention to the appropriate group of headers)

    Determine if the owner of the message file is trusted user

    if (there is no sender specified in the envelope) {
        if (there is a Sender or From field in the message header
            && the owner of the message file is trusted)
            Use the header value from it as the message sender
        else
            Generate a sender based on the owner of the file
        if (there is still no sender)
```

```
        Generate a sender referring to the local Postmaster
} else {
    if (the owner of the message file is not trusted) {
        Save message file for Postmaster to see
        Generate a sender based on the owner of the file
    }
}
if (an error occurred during parsing of the message envelope) {
    Save the message file for the Postmaster to see and correct
    return;
}
if (an error occurred during parsing of the message header) {
    Save the message file for the amusement of the Postmaster
    header_error = TRUE;
} else
    header_error = FALSE;
default address delivery uid = nobody;
if (the owner of the message file is trusted) {
    if (an incoming channel is specified in the envelope)
        set the trusted channel origin accordingly
    if (an incoming host is specified in the envelope)
        set the trusted host origin accordingly
    if (an incoming user is specified in the envelope)
        set the trusted user origin accordingly
    if (any element of the trusted origin triple is null) {
        set the resolved origin triple by
            routing the sender address
    }
    if (the message origin is a local user)
        default address delivery uid = uid of that user;
} else {
    /* We know sender is local */
    default address delivery uid = uid of owner of message file;
    if (message header contains a Sender, but no From field) {
        Rename the "Sender" field into a "From" field
    }
    if (the message header specifies a Sender) {
        if (the specified Sender address does not correspond
            to the resolved origin of the message) {
            Rename the "Sender" field a "Fake-Sender" field
            Set a flag to generate a Sender header
        }
    } else if (the message header only specifies a From field) {
        if (the specified From address does not correspond
            to the resolved origin of the message) {
            Set a flag to generate a Sender header
        }
    }
    if (flag is set that we need to generate a Sender header)
        Do so based on the owner of the message file
```

```
    }
    if (default address delivery uid != nobody
       && there is no From message header) {
       Generate one based on the envelope origin address information
    }
    /* Recipient determination */
    if (there are no recipients specified in the envelope) {
       if (header_error) {
          Reject the message with a “bad header” error
          return;
       }
       Add all the message header recipient addresses (from To, Cc,
          and Bcc headers) to the message envelope recipient list
       if (there are still no recipients specified in the envelope) {
          Reject the message with a “no recipients” error
          return;
       }
    }
    if (header_error) {
       Return the message with a “bad header” warning
       Add Illegal-Object warning headers to the message header
    }
    if (there is no To message header) {
       /* Insert the To: header lines */
       Add the list of message recipients from the message envelope
          in the message header in To headers
    }
#ifdef notdef
    rewrite all addresses in the message according to the incoming-rewriting
       rules for the originating channel.
#endif notdef
    if (hostname has been set) {
       /* Make sure Message-Id exists, for loop control */
       if (there is no Message-Id message header)
          Generate a message id and add it to the message header
       else
          extract a message id from the existing header
       Log the message id
    } else
       there is no message id
    /* Route recipient addresses */
    for (every recipient address in the message envelope)
       delivery privilege of address = default address delivery uid
    for (every recipient address in the message envelope) {
       router()   /* Route the address */
       if (the returned triple is null)
          continue;   /* ignore this recipient */
       /* Rewrite this envelope address */
       crossbar(source triple, recipient triple)
       if (the return value from crossbar() is null)
```

```
        continue;   /* ignore this recipient */
    else if (the message header rewriting function name is null) {
        Save the message for the Postmaster to see
        return;
    }
    /* Don't send message to the same address twice */
    if (we have already seen this destination triple) {
        if (the message is going to be sent to that destination)
            continue;   /* suppress duplicates */
    }
    if (this destination triple has not been alias expanded
        && it represents a local destination
        && it has an alias expansion) {
        Add the list of expanded addresses to the list of
            addresses processed by this loop, each with
            a delivery privilege determined by the source
            of the alias expansion
        continue;   /* ignore this address */
    }
    Flag that this destination triple will be sent out
    Add the address destination triple to a list for each channel
    if (the message header address rewriting function name
        is new for this message) {
        Add the name to a collection of the kinds of message
            header address rewritings that need be done
    }
}
for (every kind of message header address rewriting we need to do) {
    Call the indicated function with every address in the header
    Store the transformed headers for later use
}
if (there is no Date message header)
    Generate one based on the modification time of the message file
/* Emit specification to the transport system */
for (every recipient address) {
    if (control file has not been created) {
        Create message control file
        Write a standard preamble consisting of
            the corresponding message file name
            the offset of the start of the message body
            the message id (if any) for log identification
        if (this message did not come from the error channel)
            Write an error return address
    }
    if (the envelope sender address form for this recipient
        is different from the previous sender address form)
        Write the sender address origin triple
    Write the recipient address destination triple, and the
        corresponding address delivery privilege
    if (message header for this address is different than
```

```
            the message header for the next recipient address) {
            Write the complete message header for this destination,
                as reconstructed from the original message
                header and the stored headers transformed by
                message header address rewriting
        }
    }
    if (we created a message control file) {
        relink the message file itself to the 'QUEUE' directory
        relink the control file to the 'TRANSPORT' directory
    }
    return;
}
```

3.12 Logging

When the Router starts up as a daemon, it will attach its standard output and standard error streams to a log file. All messages from the Router will appear on one of these streams, and will therefore show up in a central location for perusal by the Postmaster or other interested parties. Usually only abnormal occurrences will be logged in this manner, but any messages printed will show up here. In particular, many of the components of the Router contain trace print statements that can be enabled at run-time. In fact, interactive debugging of the configuration file is performed this way, since when the Router is run in the foreground, the standard output and error streams are attached to the terminal in the normal fashion. Thus, all messages will appear in front of the person testing the configuration.

The tracing functionality is controlled either on the command line, or by calling the `trace` and `untrace` functions from within the configuration file, or interactively. The interactive behaviour of the Router, is to read and execute its configuration file (as normal), and then sit in an infinite loop reading commands from its standard input stream. This allows a person executing the Router interactively, to execute arbitrary statements in the configuration file programming language. The statements typed in are buffered until an End Of File indication, and then executed by the configuration file interpreter. This cycle is repeated until a syntax error occurs, or the process is interrupted. [XX: yes, this is a very rough mode of interaction. do you have any suggestions for improving it?]

The `trace` and `untrace` functions take one or more words as arguments, and turn on (off) flags that enable tracing in a component of the Router corresponding to each word. The current list of words, and the corresponding actions traced, are:

`alias` alias expansion

`all` turns all trace flags on

`assign` variable assignment

`bind` the BIND nameserver responses

`compare` case label pattern matching

`db` database lookups

`final` print message information after sequencer returns

`functions`
function calls and returns

`matched` successful case label matches

`memory` memory allocation statistics

`off` turns all trace flags off

`on` same as `functions`

`parsetree`
the configuration file parse tree

`regexp` regular expression execution

`resolv` the BIND resolver library `RES_DEBUG` option

`rewrite` message header rewriting

`router` envelope recipient address routing

`sequencer`
control flow in the sequencer function

In addition to this, each message processed is logged via the standard system logging facility (syslog) if it is available, and when not especially disabled at the router invocation. (If normal message logging to syslog is disabled, important error reporting messages are still logged.)

3.12.1 Address Testing

For example, if you wish to see how an address is routed, you can run the command:

```
echo "trace on ; router $address" | router -I
```

which, with `$address` bound to '`bond@sis.mod.uk`' might produce something like:

```
GNU Mailer router (Zmailer alpha.1 #0: Sun Jan 31 17:38:53 EST 1988)
    rayan@ephemeral.ai:/usr/src/zmailer/router
Copyright 1988 Rayan S. Zachariassen

router: parameters: 'bond@sis.mod.uk'
    echo: parameters: 'router:' 'bond@sis.mod.uk'
router: bond@sis.mod.uk
    echo: returns: ''
    canonicalize: parameters: 'bond@sis.mod.uk'
        focus: parameters: 'bond<@sis.mod.uk>'
            [: parameters: 'sis.mod.uk' ']'
            [: returns: 'true'
        focus: returns: 'bond<@sis.mod.uk>'
    canonicalize: returns: 'bond<@sis.mod.uk>'
    [: parameters: '' ']'
    [: returns: ''
    [: parameters: 'sis.mod.uk' '==' 'ephemeral.ai.toronto.edu' ']'
    [: returns: ''
    [: parameters: 'sis.mod.uk' ']'
    [: returns: 'true'
router: returns: 'smtp' 'sis.mod.uk' 'bond@sis.mod.uk'
```

4. Scheduler

The Scheduler complements the Router as the other major process in ZMailer. The decisions it makes involve how to manage and time delivery of messages to their destination, and its name arises from this scheduling function. While the Router interprets message files, the Scheduler interprets only the control files corresponding to the message files.

The control files are usually produced by the Router, and appear in a directory scanned by the Scheduler daemon. Whenever a new control file does appears in that directory, its contents are used to update a data structure, maintained by the Scheduler, that describes which addresses in which messages are destined for which hosts and channels. The information stored along with each channel/host combination is a set of byte offsets into the control file, giving the location of address specifications corresponding to that combination. This information can later be passed to a transport/delivery program, and is updated based on feedback from these programs.

A Transport Agent is a program that the Scheduler executes to deliver messages. The Scheduler determines the correspondence between channels, hosts, or channel/host destinations, and a specific Transport Agent, by interpreting a simple table from a configuration file. The Transport Agent process is told which control files it should inspect for work, and tells the Scheduler the status of the destination addresses it tried to process. The Scheduler then updates the model it maintains of work that needs to be done, and will eventually remove the last link to a control file and its corresponding message file. At that point, ZMailer has done its job with regard to each message.

Note that the communication between the Scheduler and other programs is mostly via the message control file, instead of by direct interaction (when the Scheduler converses with Transport Agents).

4.1 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.

4.1.1 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 optional semaphore data (transporter process id, for example), followed by the field value.

Here is a simple control file produced by a test message, just before it was removed by the Scheduler:

```
--------------------
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:

```
#define _CF_MESSAGEID  'i' /* inode number of file containing message */
#define _CF_BODYOFFSET 'o' /* byte offset into message file of body */
#define _CF_BODYFILE   'b' /* alternate message file for new body */
#define _CF_SENDER     's' /* sender triple (channel, host, user) */
#define _CF_RECIPIENT  'r' /* recipient n-tuple, n >= 3 */
#define _CF_ERRORADDR  'e' /* return address for error messages */
#define _CF_XORECIPIENT 'R'/* one of XOR set of recipient n-tuples */
#define _CF_RCPTNOTARY 'N' /* DSN parameters for previous recipient */
#define _CF_DSNENVID   'n' /* DNS 'MAIL FROM<> ENVID=XXXX' data */
#define _CF_ERRORADDR  'e' /* return address for error messages */
#define _CF_DIAGNOSTIC 'd' /* diagnostic message for ctlfile offset */
#define _CF_MSGHEADERS 'm' /* message header for preceeding recipients */
#define _CF_LOGIDENT   'l' /* identification string for log entries */
#define _CF_OBSOLETES  'x' /* message id of message obsoleted by this */
#define _CF_VERBOSE    'v' /* log file name for verbose log (mail -v) */
#define _CF_TURNME     'T' /* trigger scheduler to attempt delivery now */
```

There is one field per line, except for `_CF_MSGHEADERS` which has some special semantics described below. The following describes the fields in detail:

`i` This field identifies the message file corresponding to this control file. It is the name of the message file in the '`QUEUE`' directory ('`~/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 the destination addresses have been processed. For example:

```
i 21456
```

`o` 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
```

`e` Gives an address to which delivery errors should be sent. The address must be an RFC822 *mailbox*. For example:

```
e "Operations Directorate" <d-ops@sis.mod.uk>
```

`l` 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>
```

`s` 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` 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 example:

```
r local - bond 0
r uucp uunet sisops!bond -2
```

`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!
```

`d` This field is *not* written by the Router. It is written by the Scheduler 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 [XX:or at other times]. For example:

```
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.

4.2 Scheduler Configuration File

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. A typical configuration file would look something like:

```
# pattern    intvl   ch/ho/* uid     gid     command
local/*      10s      2 0 0  root    daemon  mailbox local
smtp/*       1m      10 2 0  root    daemon  smtp -l /tmp/smtp.log $host
error/*      5m      10 0 0  root    daemon  errormail
uucp/*       10m     10 0 0  root    daemon  sm -c $channel uucp
```

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. Each line consists of eight white-space separated fields. The fields, in sequence, are:

A pattern, that selects which channel/host combinations are relevant to the current line. The pattern has the form: *channel/host*, with the slash being mandatory. The subpatterns (i.e. each side of the slash) may contain a `glob` (or `sh`) style pattern. These patterns are tested in the order they appear, with the channel and host values for destination addresses in a message. When both patterns match, the line with the matching pattern describes the Transport Agent that should be used to deliver the message to that destination. It is important that the Transport Agent recognizes at least the set of addresses the in message control file, that the Scheduler configuration table assumes it does. Otherwise, some addresses may never get delivered to, and the message will stay in the Scheduler indefinitely.

An interval specification that says how often the Scheduler should check for work pending for the Transport Agent described by that line. The time specification must use an appropriate suffix: '`s`' for seconds, '`m`' for minutes, '`h`' for hours, or in combinations, e.g. '`1h30m`'. The minimum value specified in the configuration file will be the directory scanning interval used by the Scheduler.

A maximum number of Transport Agents simultaneously active for the channel matched by the pattern for that entry. If 0, no upper limit is enforced.

A maximum number of Transport Agents simultaneously active for the host matched by the pattern for that entry. If 0, no upper limit is enforced.

A total maximum number of Transport Agents simultaneously active due to that entry. If 0, no upper limit is enforced.

A user id to set as the real and effective user id when executing the command associated with that entry. Either a symbolic (login name) or numeric value may be specified.

A group id to set as the real and effective group id when executing the command associated with that entry. Either a symbolic (login name) or numeric value may be specified.

Finally, the Transport Agent invocation command itself, as it would appear on a normal command line. Note however that the Scheduler executes the command directly without all the command line interpretation afforded by a shell. The only special action is to replace instances of the word '`$channel`' with the name of the channel matched by the pattern, and instances of '`$host`' with the name of the host matched by the pattern.

Note that the command must have enough privileges specified to write into the control file, in addition to whatever is necessary to perform its delivery duties and logging.

4.3 Transport Agent protocol

Once the Scheduler starts up a Transport Agent by executing one of the commands specified in the configuration file, it needs to pass information to the Transport Agent about which messages and addresses it should process. The Transport Agent in return needs to report to the Scheduler about the success or failure of its delivery activity, so that issues related to file management and error reporting can all be centralized in the Scheduler process.

To accomplish this, the Scheduler engages in a simple exchange with each Transport Agent it has started. For this reason, the Scheduler creates two pipes attached to the standard input and standard output of the Transport Agent processes it executes. The standard error descriptor is shared with the Scheduler process, and usually refers to the Scheduler log file.

Just before the Scheduler starts up a Transport Agent, it scans through its model of the pending message control files, and determines which are relevant to the impending invocation of the Transport Agent. Once the subprocess is running, the Scheduler will write to the standard input of its child, the names of the control files it should process, one to each line. This list is terminated by an empty line, to indicate to the Transport Agent that the Scheduler finished its business normally.

In turn, the delivery process will open each named control file, scan it for destination addresses relevant to its specific invocation, and attempt delivery to those addresses. For each destination address, it will print to its standard output a line that describes the address, and that contains a status indication. The syntax is:

id/offset/status comment

The *id* is the message file id contained in the '`i`' field of the control file. The *offset* is the byte offset into the pertinent control file of the destination address field. The *status* is one of the following list of keywords understood by the Scheduler:

- `ok` Delivery was successful.
- `error` Delivery was unsuccessful.
- `deferred` Delivery was attempted, but is deferred.

The optional *comment* is an arbitrary string that clarifies the *status* code. It is separated from the *status* code by a single space. For example, the following is a possible sequence of reports:

```
18453/3527/ok
18453/3565/deferred Unable to contact sis.mod.uk!
18453/4211/error No such local user: 'bond'.
```

After each message file has been processed, an empty line is output to indicate this. The Transport Agent will continue to the next message control file (if any) that has been written to it by the Scheduler.

A complete exchange between a Scheduler and a Transport Agent might proceed as shown:

```
Scheduler          Transport Agent
----------------------------------------------------------------------
smtp/21456
                   18453/878/ok
                   18453/1013/error Illegal hostname: 'spectre'
                   _
local/21456
                   18453/945/deferred Cannot lock mailbox: 'bond'
                   _
_
```

The underscores indicate empty lines emitted by either side in the synchronous protocol. After such a conversation, the Transport Agent process will exit gracefully. Whenever the status of a destination is updated, the Scheduler will check its internal data on whether or not a link to the control file should be removed, or if indeed delivery has been completed and both the message file and the last links to its control file should be removed.

4.4 Mail Queue printing

It is possible to read through the directory hierarchy under the 'TRANSPORT' directory to synthesize a model of which messages are queued to go where. However, this method does not guarantee an accurate image of the model within the Scheduler process, nor can it provide any status information (as given by the status commentary of Transport Agents) other than success or failure. The ideal solution would be a way of interrogating the Scheduler itself about the current state, and then perhaps use this as a basis for verbose embellishments.

Such a facility is incorporated in the Scheduler. The exact interrogation mechanism depends on the facilities of the host operating system: a system with TCP/IP would use a socket rendezvous, a system with named pipes would use a prearranged special file and signalling mechanism, a system without either would rely on normal files.

In all cases, the result of the interrogation is a terse list of of messages, their destinations (channel/host combination), and the offsets of the addresses corresponding to each destination. For example, a sample state dump from the Scheduler is:

```
29198:  smtp/csri.toronto.edu, 2 addresses [196,247]
        smtp/ephemeral.ai.toronto.edu, 1 address [128]
```

This shows one message (id 29198) queued for transmission via SMTP to two different destination hosts. One destination has two associated addresses, referred to by byte offsets (196 and 247) into the control file for the message ('~/transport/29198'). The other destination has only one address associated with it.

The above dump corresponds to the state just after the Scheduler has parsed the control files. After the Transport Agent corresponding to the smtp/ephemeral.ai.toronto.edu destination has exited, the state might become:

```
29198:  smtp/csri.toronto.edu, 2 addresses [196,247]
        smtp/ephemeral.ai.toronto.edu, 1 address [128]
                connect: Connection refused (will retry)
```

In this case, the host ephemeral.ai.toronto.edu is alive, but no SMTP server is running. After the Transport Agent for the other destination has exited, the state might be:

```
29198:  smtp/ephemeral.ai.toronto.edu, 1 address [128]
                connect: Connection refused (will retry)
```

Finally, once this remaining destination is processed successfully, the Scheduler reports:

```
Mail queue is empty
```

These reports from the Scheduler succinctly express the state of the queues in a format that is human-readable, and that is also easy to parse automatically. The only information not provided are the actual addresses referred to by the state dump. The program that queries the Scheduler for this information is capable of finding these addresses if it needs to (assuming the control files are readable), and presenting it in a different format. The Scheduler does not remember the actual address information, and so cannot easily include it in the dump. Since the Scheduler must spend a minimum of time servicing requests from Transport Agents and mail queue queries, it leaves nontrivial work to the querying program.

Some advantages arise from this mechanism: in environments with host-to-host interprocess communication (e.g. TCP/IP) it becomes possible to query Schedulers on remote hosts about their state, and such remote queries can only get verbose information if the querying process has access to the control files of the remote ZMailer installation. This makes it possible for an environment making use of distributed filesystems to have a single ZMailer installation on a mail server host, and for all the other local machines to access its services transparently. At the same time, no private information can be divulged without direct access to the '`POSTOFFICE`' directory.

The ZMailer distribution contains a utility program `mailq` that is used to query Schedulers. It supports the transparency paradigm in an NFS environment, by arranging to query the Scheduler running on the NFS server host for the '`POSTOFFICE`' directory visible on the local host.

4.5 Logging

As with the Router, the Scheduler daemon will attach its standard output and standard error streams to a log file. The standard error stream of each Transport Agent invocation is inherited from the Scheduler, and so is attached to the same log file. The Scheduler does have an option to produce a debugging log, but otherwise only extraordinary occurrances are logged (for example, a delivery failure, missing Transport Agents, etc.).

5. Transport Agents

A Transport Agent is a program that delivers mail to a particular destination. The destination paradigm in ZMailer involves the concept of a channel, a next-host, and a next-address. The first two are used by the Scheduler to select a Transport Agent, and by a Transport Agent to identify which destinations it should process. Any necessary information to accomplish this selection, is either contained within the Transport Agent, or supplied by the Scheduler on the command line when invoking a Transport Agent. The message control files examined by a Transport Agent instance are passed by the Scheduler in a simple protocol designed for the purpose, and status reports on the actions of the Transport Agent are returned by the same protocol.

When a Transport Agent starts up, it expects to read message control file path names on its standard input stream, and will print status reports on its standard output stream. Unexpected errors are sent to the standard error stream for logging. A Transport Agent can be invoked interactively for test purposes, but usually it is started as a child of the Scheduler daemon, with its input and output streams attached to the Scheduler using pipes, and sharing the error stream with the Scheduler itself (and other concurrent Transport Agent processes).

The following sections describe the Transport Agent programs that come with the ZMailer distribution.

5.1 Local delivery (mailbox)

The delivery of local mail is of paramount importance in a mailer. Of all the things that might go wrong during mail processing, a mistake by the local delivery process can be the most critical. Since it is also a very frequent operation, this Transport Agent must be both robust and efficient. Perfection is elusive, but the local delivery program included with ZMailer has proven itself in the original version used with Sendmail.

This program will look for destinations with a channel of '`local`', and will ignore the next-host specification. The next-address specification is either a local account id, a full path to a file, or a pipe ('`|`') followed by a valid argument to `sh -c`. The following are examples of *legal* values:

```
bond
/usr/arch/lists/info-widget
/etc/passwd
|sed -e '1,/^$/d' >> /etc/passwd
|/bin/mail badhost!badguy </etc/passwd
```

Here are some *illegal* variations:

```
<bond>                              (angle brackets invalid in local-part)
"bond"                              (double quotes unlikely in login id)
bond@sis.mod.uk                     (local-parts do not contain '@')
james bond                          (whitespace unlikely in login id)
lists/info-widget                   (not an absolute pathname)
sed -e '1,/^$/d' >> /etc/passwd     (does not start with a '|')
```

Note that the effect of these addresses depends on whether the local delivery program actually honours the request, and if it does which privileges are used while executing the indicated action.

Specifically, if the next-address does not start with either '`|`' or '`/`', it is assumed to be a user name. This is checked by lookup in the system account database ('`/etc/passwd`'), to determine which user id should own the mailbox file. If the indicated account exists, mail is delivered to its corresponding mail spool file, in the standard format (return address and delivery date in a '`From `' line preceding the actual message, etc.). Note that even though account names should ideally be case-independent there is no portable way of doing a case-independent lookup in the account database. The Router deals with this by first trying the name lookup without doing a case translation, and if that fails the uppercase letters in the *local-part* will be mapped to lowercase and the lookup retried. The local delivery program does *no* aliasing on its own.

If delivery to a file or command is indicated, the actual delivery is done using the user id listed as the destination address privilege in the control file. What this actual privilege allows, is up to the security mechanism in the Router. Since addresses specified from a remote host start out with minimal privileges, they will usually not cause any harm on the local system.

Programs executed by this Transport Agent will be given an environment containing the `$PATH`, `$SHELL`, `$HOME`, `$USER`, `$UID`, and `$SENDER` environment variables. The first two are constant, the next three depend on the delivery privilege of the address, and the value of the last environment variable is set to be the return address of the message being delivered. The current directory is set to be `$HOME` when possible.

The local delivery program contains code that may be enabled at compile time to honour the `comsat` protocol. There are separate symbols to enable this for local users (`BIFF`) and for remote users (`RBIFF`). In the former case, users would enable the feature individually by executing `biff y`, while in the latter case a '`.rbiff`' file in the user's home directory triggers the remote notification of new mail.

5.2 Error mail delivery (errormail)

The error messages of ZMailer are stored as message file forms in a specific '`FORMS`' directory. The various ZMailer programs will access the appropriate forms directly, but errors detected in the Router configuration file must be handled in a different way. By convention, any problem found by configuration file code is handled by changing the message destination to be a triple of the form:

(`error`, *form*, *address*)

The '`error`' channel is serviced by this Transport Agent, which expects the *form* listed to be the name of a file in the '`FORMS`' directory. This should be a prototype message file, containing all generic information associated with the error (i.e. the message header lines and an appropriate explanation to the user). The *address* is the address rejected by the configuration file code, for whatever reason is given in the *form* file.

By convention, the names of the *form* files indicate the class of error that occurred. The following describes the standard forms that come with ZMailer:

'`err.badheader`'
: Syntax error in the message header.

'`err.delivery`'
: Delivery problem, used by the Scheduler on behalf of Transport Agents.

'`err.nonewsgroup`'
: A non-existent USENET Newsgroup was addressed.

'`err.norecipients`'
: The message has no recipients listed.

'`err.unresolvable`'
: The routing code in the Router configuration file cannot determine a destination for the message.

'`warn.badheader`'
: Used to chastise a user who sends improperly formatted mail.

Of these, only '`err.nonewsgroup`' and '`err.unresolvable`' are referred to by the Router configuration file, the rest are used internally by the Router or Scheduler. Therefore these forms *must* be available for proper operation of ZMailer.

To illustrate, here is the default '`err.badheader`' form:

```
--------------------
From:   The Post Office <postmaster>
Subject: Invalid message header
Cc:     The Postmaster <postmaster>

The following message arrived with an illegal header according to the
RFC822/976 protocol specification. If you do not recognize the source
of the bad header, perhaps you should ask a postmaster at your site.

The following annotated headers illustrate where the error(s) occurred:

--------------------
```

5.3 SMTP client (smtp)

The SMTP Transport Agent implements this message transfer protocol according to RFC821. It scans message control files for a channel called '`smtp`', and a next-host as specified on the command line. Only a single virtual circuit (VC) is established to the remote SMTP server, and all transactions are carried out in sequence across this VC. By contrast, Sendmail opens a new VC for every mail message.

This program does not enforce the line length limits of the SMTP protocol, nor does it check that the message file data is 7 bit ASCII. However, the CRLF line termination rule is followed, as are all other aspects of the SMTP protocol. When connected to a ZMailer SMTP server program, message bodies containing arbitrary binary data may be transferred (since the SMTP DATA encoding is reversible, and there are no line length limits on either end).

A log file may be specified for recording the SMTP transaction.

5.4 Sendmail compatible delivery programs (sm)

Because Sendmail already has many "mailer"s written for it, and to ease the transition from Sendmail to ZMailer, this Transport Agent was written to interface with such programs from the ZMailer environment. The basic characteristic of a Sendmail "mailer" is that its command line specifies what must be done with the message available on its standard input stream.

Because of the generic interface, this Transport Agent requires a small configuration file which it reads on startup. The configuration file declares which programs are available, how to invoke them, and what channel each program corresponds to. Here is a sample configuration file:

```
# M     F =    P =                      A =
local   mS     /usr/lib/mail/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)
news    m      /usr/lib/mail/pnews      post.news $h $u
```

The configuration file is a table with each line containing four fields: a channel name, Sendmail "mailer" flags, the full path name of the program to execute, and the command line that program should see.

The flags field contains the flags that are appropriate to the ZMailer environment, for example the presently recognized flags are:

- `f` — Include a '`-f` *sender*' in the command line.
- `r` — Include a '`-r` *sender*' in the command line.
- `S` — Do not reset the uid to the real uid of the Transport Agent process.
- `n` — Do *not* prepend a '`From `' line to the message.
- `s` — Strip quotes on addresses [XX:todo].
- `m` — Many recipients may be handled by a single instance of the command.
- `P` — Add a "Return-Path" message header.
- `U` — Prepend a '`From ... remote from ...`' line to the message.
- `X` — Use the SMTP hidden dot algorithm (i.e. escape periods on a line by themselves).
- `E` — Replace occurrences of '`From `' at the start of a line in the message body with '`>From `'.
- `7` — Pass 7-bit ASCII by stripping 8th bit of bytes in the message [XX:todo].
- `-` — No-op flag.

Mailer flags that are not mentioned in the above table have been excluded due to their lack of semantics in this situation. Typically their functionality should be accomplished in the Router instead [XX: if it isn't, and it is needed, please let me know].

The command line specification may contain anything valid in the same field in a Sendmail "mailer" definition. In particular, any argument containing '`$u`' is expanded as many times as there are recipients that can be dealt with at once by that command. The '`$g`' macro expands to the return address of the message, and '`$h`' to the next-host in the destination.

At present, no special environment is set up for programs executed by this Transport Agent. The standard output and standard error of such processes are caught by the Transport Agent, and the first line read (if any) is passed on to the Scheduler using the normal status reporting mechanism.

6. Miscellaneous

6.1 Sendmail compatibility

After installing the Sendmail compatible ZMailer interface programs, the present user-visible incompatibilities with Sendmail proper are:

- Verbose mode (‘`-v`’ flag to Sendmail) is not implemented.
- Occurrences of ‘`:include:`’ specifications in the aliases database must be quoted.
- The “Return-Receipt-To” message header is not yet honoured.
- The mailing-list management features of Sendmail are not implemented, avaiting consultation.

6.2 SMTP server

The ZMailer distribution contains an SMTP server program for the BSD socket implementation of TCP/IP. It is an asynchronous implementation, in that address semantics are not checked in real time, nor are other (optional in the SMTP standard) functions that require Router functionality. The server simply says “Yes yes, sure!” to everything, and passes the information to the Router for verification. The program may also be used in non-daemon mode to unpack BSMTP format messages on the standard input stream. For compatibility with the Sendmail variation on the SMTP protocol, it accepts the `VERB` and `ONEX` commands as No-Ops. The `VRFY`, `EXPN`, `HELP`, and `TURN` commands are presently unimplemented, as is the case for the interactive `SEND`, `SAML`, and `SOML` commands.

7. How-To Guide

This chapter is intended to give practical tips on topics related to the maintenance and customization of ZMailer. If you want to see something covered here, let me know.

7.1 How to install ZMailer

Thie 'README' file in the distribution contains specific instructions for installing ZMailer. The following goes into slightly more depth than the 'README' file does:

The documentation for ZMailer (part of which you are reading right now) is maintained in *texinfo* format. To format this for a high-quality output device requires that you already have TEX running, and that you have the Texinfo macro package installed in the TEX macro library. If not, these macros are part of every GNU Emacs distribution (and included with the current ZMailer distribution). Generating a line printer or screen version of the documentation requires the aid of GNU Emacs (see 'doc/Makefile'). If you have neither TEX nor GNU Emacs, ask me for a preformatted version of the documentation.

There is very little hardcoded configuration information in the ZMailer programs. The 'conf.c' files in the 'router' and 'scheduler' subdirectories of the distribution are the primary locations of static configuration information. You should check these files, but there is no need to change them unless you know what you are doing, and insist.

The only other static global information is kept in the 'mail.h' header file in the 'include' subdirectory. In an operating system environment that integrates ZMailer, this file is intended to go in '/usr/include'.

There is some dynamic global information, and other compile time information, that needs to be specified somewhere. The way it is done with ZMailer, is that you (the installer) edits a global configuration file ('Config'), which contains variable definitions that will propagate to all the makefiles in the distribution. These definitions will also appear in a file '/etc/zmailer.conf' that ZMailer programs refer to for global information. This information includes for example the locations of the 'POSTOFFICE' directory hierarchy, so this facility allows easy dynamic reconfiguration of some installation parameters. The file is in '/etc' to increase reconfiguration flexibility for diskless mail clients.

Canned error and warning messages are kept in the 'proto/forms' directory of the distribution.

They should be modified to suit local preferences. By default, all errors will be carbon-copied to the postmaster, a local address that is hopefully defined in the aliases database. Until you are comfortable with the ZMailer system, you should probably use the default forms.

Once the above preliminaries have been taken care of, the time has come for your computer to earn its keep. If you run the command:

```
make it so
```

the following will happen:

- A recursive `make clean` is run to scrub the distribution hierarchy.
- The global `make` file ('`Makefile`') is edited to update it with rules for updating all the `make` files in the distribution when the '`Config`' is modified.
- The '`Config`' file is processed into a `sed` script, which is then applied to all the `make` files in the directory tree.
- All programs are compiled.
- Another `sed` script constructed from the '`Config`' file is applied to update the '`proto/zmailer`' shell script.
- The '`POSTOFFICE`' directory hierarchy is created, and the canned error messages from the '`proto/forms`' directory are copied to '`~/forms`'.
- The ZMailer directory hierarchy under '`/usr/lib`' is created, and the standard configuration and control files and shell scripts from the '`proto`' directory are copied to that location (referred to as '`MAILBIN`' in the '`Config`' file).
- All the program binaries are installed under the '`$MAILBIN`' directory.
- Finally, another `sed` script is applied to the '`Config`' file, to produce '`/etc/zmailer.conf`'.

Then it is time to get your aliases database working. If you don't already have a central aliases database, you should create one. The minimum requirement is that the `postmaster` address expands to a real account id. If you already have a central aliases database, this is typically because you are currently running Sendmail. In that case, start by copying the Sendmail aliases file to '`$MAILBIN/db/aliases`'. The ZMailer Router is used to build the aliases database from the aliases file. To do this correctly, the Router must know what kind of aliases database to access and, in this situation, create. The distribution Router configuration file will check for the existence of a '`$MAILBIN/db/aliases.dir`' file to indicate that NDBM or DBM is being used. If this file is absent, the Router will access the database using a binary search algorithm on an index file.

If you are using NDBM, prepare the way for the Router by creating a null '`aliases.dir`' file in the 'db' subdirectory. Then run the Router to initialize the aliases database (`router -i`). If you get syntax errors, correct them in the 'aliases' file. Eventually the Router will report some simple counts (a la Sendmail) of defined aliases, indicating it was successful in initializing the aliases database.

You should now arrange for host-specific information to be made available to ZMailer. This is obviously a very site-specific customization. Although the method of access and location of such information is defined in the Router configuration file (which incidentally is '$MAILBIN/router.cf'), certain Transport Agents need to know the hosts' UUCP node name. This is read from the file '/etc/uucpname' if it exists, and secondarily obtained from the `uname` system call in certain environments. The convention of using '/etc/uucpname' is due to 4.3BSD UUCP which allows this as a configuration option. I recommend this method, since it greatly increases portability of the UUCP binaries between your machines.

The sample Router configuration file in the distribution, assumes that the host names is should deliver mail locally for, are listed in the file '$MAILBIN/db/localdelivery'. For example, in an environment with a mail server and clients, all hostnames should be listed in this file. This is suggested as a convention for how to discover this information, and where. The sample configuration file should be studied for other guidance of this sort.

You can finally try running the Router in interactive mode, as illustrated in the 'README' file. This is the stage at which you should start playing with the configuration file and with the various ZMailer programs. This is also an opportune moment (or day) for you to customize or write a Router configuration file for your host/site.

If you have an '/etc/services' file, it should be updated with the definition of a TCP port used for mail queue querying. The Scheduler acts as a server listening on this port, and the `mailq` program included with the distribution will connect with the Scheduler and obtain a dump of the mail queue by this mechanism. If your system does not have TCP, a rendezvous mechanism using named pipes will be used instead. For systems without either of these facilities, a prearranged file is used along with a release protocol when the mail queue dump is completed.

When you are comfortable with the new environment and want to start ZMailer, there is a shell script provided ('$MAILBIN/zmailer') to carry out the normal startup functions. If invoked without arguments, it will start the Router and Scheduler daemon processes, and the SMTP server process. The latter may clash with any running Sendmail daemon if it is also acting as an SMTP server. This script may also be invoked with individual arguments like `router` or `scheduler` to start up just the specified process(es). It may be run from the '/etc/rc.local' file to start ZMailer on reboot.

7.2 How to write a Router configuration file

Sorry, I don't know what the problem areas will be at this point, so this section is incomplete. The following is a quick summary:

The configuration file is read and all statements executed sequentially. Like any other statement, a function definition is also executed, with the side effect of defining a function. All functions must be defined before use. Normal statements appearing at the top level in the configuration file (i.e. not within a function definition), usually have the purpose of setting up an environment for the rest of the configuration file. An "environment" encompasses global variables (e.g. "what is my name") initialized in assignment statements, and database definitions by the `relation` statement.

There are four instances of magic semantics assumed by the Router:

- Setting the hostname by calling the `hostname` function, will enable generation of trace headers (i.e. '`Received`' and '`Message-Id`').
- The aliases database is defined by the `aliases` relation. The value of a database lookup must be a byte offset into another file that contains the actual alias definitions.
- A `router` function must exist. It takes an address as its one argument, and returns three values representing the channel, next-host, and next-address.
- A `crossbar` function must exist. It takes two triples (six arguments) and returns those triples and the name of a rewriting function to be applied to all the header addresses (seven values). The argument triples represent the origin and recipient envelope addresses, and this function is in charge of rewriting them as appropriate.

Naturally, all function names returned by the `crossbar` function must correspond to a defined function.

Tell me what is missing from this description. Would a play-by-play of the sample configuration file be very useful?

Appendix A. User Agent support

A.1 Submission Interface

Three C library routines are provided to open (create), abort (remove), and close (submit) a message file. Internally, they make use of the stdio package, and their interface is modelled after it. The interface definition is:

```
#include <mail.h>

FILE *mail_open()

int mail_abort(mfp)
FILE *mfp;

int mail_close(mfp)
FILE *mfp;
```

The parameter passed in a `mail_abort()` or `mail_close()` call is the value returned by a call to the `mail_open()` function. The routines take care of all the necessary housekeeping. They are properly used as follows:

```
...
FILE *mfp;
on exit or interrupt, arrange to call mail_abort(mfp);
if ((mfp = mail_open()) == NULL)
    ... error handling when message submission is not possible ...
 else
    ... output the mail message to mfp ...
    if (oops && (mail_abort(mfp) == EOF))
        ... print a message that the abort failed ...
    else if (mail_close(mfp) == EOF)
        ... error handling when message submission fails ...

reset behaviour on exit or interrupt
...
char *tmalloc(n) unsigned int n;  return n bytes of memory
```

Notice the definition of `tmalloc()`. This routine should allocate memory that will remain usable within the lifetime of the message submission (i.e. until a `mail_abort()` or `mail_close()` call). This allows a User Agent or other application program that makes many calls to these routines during its lifetime, to provide an alternate byte allocator that will not cause them to run out of data space.

Another point to be made is that these routines and all other code in ZMailer that relinks files, uses `link()`/`unlink()` combinations and never the `rename()` system call, even if it is available. Unfortunately, `rename()` does not retain the inode number of the file being renamed.

Finally, although this interface will honour the **FULLNAME** and **PRETTYLOGIN** environment variables mentioned earlier, a User Agent can override this mechanism by seeking to byte 0 of the message file and writing its message data from there.

The system standard header file '`mail.h`', declares these routines appropriately. It contains all the common definitions used in passing information between the components of ZMailer. This includes the names of various directories, the postmaster, and symbolic names for various keys used in the control file protocol.

A.2 Fullname quoting

The library routine that constructs a full user name, does so purely based on information passed to it. This means it can be used with the contents of a GECOS field (everything after a '`,`' or a '`;`' is ignored), or some other arbitrary string, without incurring any unnecessary cost involved in a password database lookup. The interface specification is as follows:

```
char *
fullname(gecos, buf, buflen, login)
        char *gecos;            /* the name we wish to quotify */
        char buf[];             /* place to put the result */
        int buflen;             /* how much space we have */
        char *login;            /* what to use for a login name */
```

The return value from `fullname()` is always the value of the second parameter. A sample usage would be:

```
struct passwd *pw;
char buffer[BUFSIZ], *name;
extern char *fullname();

name = fullname(pw->pw_gecos, buffer, sizeof buffer, pw->pw_name);
```

If the fourth parameter is `(char *)NULL`, the `fullname()` routine will look for the **USER** and **LOGNAME** environment variables, in that order, if it needs a login name due to the expansion of a '`&`' in the GECOS field. For example:

```
fullname("& Kirk", ..., "jim") returns "Jim Kirk".
fullname("James T. &", ..., "kirk") returns "\"James T. Kirk\"".
```

The routine will truncate the text of its return value to fit in the space available in the buffer. If there is a leading double-quote, there will also be a trailing double-quote. The decision to quote is made according to the specifications in RFC822 for a *phrase*. In other words, when scanned according to the lexical rules of RFC822, the return value from `fullname()` will constitute a valid RFC822 *phrase*.

Table of Contents