
3

Exim overview

In the previous chapter, the job of an MTA is described in general terms. In this chapter, we explain how Exim is organized to do this job, and the overall way in which it operates. Then in the next chapter, we cover the basics of Exim administration before launching into more details about its configuration.

3.1 Exim philosophy

Exim is designed for use on a network where most messages can be delivered at the first attempt. This is usually true for most of the Internet. Measurements taken in the author's environment (a British university) indicate that well over 90% of messages are delivered almost immediately under normal conditions. This means that there is no need for an elaborate centralized queuing mechanism through which all messages pass. When a message arrives, an immediate delivery attempt is likely to be successful; only for a small percentage of messages is it necessary to implement a store-and-forward mechanism.¹

Therefore, although it is possible to configure Exim otherwise, the normal action is to try an immediate delivery as soon as a message is received. In many cases this is successful, and nothing more is needed to process the message. Nevertheless, some precautions must be taken to avoid system overload in times of stress. For example, if the system load rises above some threshold, or if there are a large number of simultaneous incoming SMTP connections, immediate delivery may be temporarily disabled. In these events, incoming messages wait on Exim's queue and are delivered later.

All operations are performed by a single Exim binary, which operates in different ways that depend on the arguments with which it is called. Although receiving and delivering messages are treated as entirely separate operations, the code for determining how to deliver to a specific address is required in both cases, because during message reception, addresses are verified by checking whether it would be possible to deliver to them. For example, Exim verifies a remote sender address by looking up the domain in the DNS in exactly the same way as when setting up a delivery to that address.

¹ Nowadays, some servers handle very large numbers of messages. This means that, although the percentage of messages that have to be retried is still small, the absolute numbers are much larger than they were when Exim was designed.

On a system where Exim is fully installed as a replacement for Sendmail, one or both of the paths `/usr/lib/sendmail` or `/usr/sbin/sendmail` is a symbolic link to the Exim binary.¹ Therefore, any MUA, program, or script that is running on the Exim host which tries to send a message by calling Sendmail actually calls Exim.

3.2 Exim's queue

The word *queue* is used for the set of messages that Exim has under its control at any one time, because this word is common in the context of mail transfer. However, Exim's queue is normally treated as a collection of messages with no implied ordering, more like a “pool” than a “queue”. Furthermore, Exim does not maintain separate queues for different domains or different remote hosts. There is just a single, unordered collection of messages awaiting delivery, each of which may have several recipients. If you are an Exim administrator, you can list the messages on the queue by running the command:

```
exim -bp
```

assuming that your path is set up to contain the directory where the Exim binary is located. Exim administrators are referred to as “admin users” (☞ 22.4.2). Messages that are not delivered immediately on arrival are picked up later by *queue runner* processes that scan the entire queue and start a delivery process for each message in turn.

3.3 Receiving and delivering messages

Message reception and message delivery are two entirely separate operations in Exim, and their only connection is that Exim normally tries to deliver a message as soon as it has received it. Receiving a message consists of writing it to local spool files (“putting it on the queue”) and checking that the files have been successfully written before acknowledging receipt to the sending host or local process. There is only one copy of each message, however many recipients it has, and the collection of spool files *is* the queue; there are no additional files or in-memory lists of messages.

All the data about the state of a message is kept in its spool files. Each attempt at delivery causes every undelivered recipient address to be processed afresh. Exim does alias, forwarding, and mailing list expansion for local addresses (where configured) and domain lookups for remote addresses every time it handles a message. It does not normally retain previous alias, forwarding, or mailing list expansions from one delivery attempt to another.

¹ Linux and BSD-based systems tend to use `/usr/sbin/sendmail`, whereas Solaris uses `/usr/lib/sendmail`. Different MUAs have different defaults, so some administrators set both paths to be on the safe side.

However, there is one exception to this: if the `one_time` option is set for a mailing list, the list's addresses are added to the original list of recipients at the first delivery attempt, and no re-expansion occurs at subsequent attempts (⇒ 5.3.3).

3.4 Exim processes

Parallelism is obtained by the use of multiple processes, but one important aspect of Exim's design is that there is no central process that has overall responsibility for coordinating Exim's actions. Therefore, there is no concept of starting or stopping Exim as a whole. Exim processes can be started at any time by other processes. For example, user agents are always able to start Exim processes in order to send messages. Such processes perform a single task and then exit. Almost all Exim processes are short-lived, but Exim does make use of long-running daemon processes for two purposes:

- (1) To listen on the SMTP port for incoming TCP/IP connections. On receiving such a connection, the listener forks a new process to deal with it. An upper limit to the number of simultaneously active SMTP reception processes can be set. When the limit is reached, additional SMTP connections are refused.
- (2) To start up queue runner processes at fixed intervals. These scan the pool of waiting messages (by default in an arbitrary order) and initiate fresh delivery attempts. A message may be on the queue because a previous delivery attempt failed, or because no delivery attempt was initiated when the message was received. Each delivery attempt processes a single message and runs in its own process. The queue runner waits for this process to finish before moving on to the next message. A limit may be set for the number of simultaneously active queue runner processes run by a daemon.

A single daemon process can be used to perform both these functions, and this is the most common configuration. However, it is possible to run Exim without using a daemon at all; *inetd* can be used to accept incoming SMTP connections and start up an Exim process for each one, and queue runner processes can be started by *cron* or some other means. However, in these cases Exim has no control over how many such processes are run, so if you are worried about system overload, you must control the number of processes yourself.¹

3.5 Coordination between processes

Processes for receiving and delivering messages are for the most part entirely independent. The small amount of coordination that is necessary is achieved by sharing files. Minimizing synchronization and serialization requirements between processes

¹ *xinetd* (www.xinetd.org) is a replacement for *inetd* that includes additional control facilities.

helps Exim to scale well. Apart from the messages themselves, the shared data consists of a number of files containing “hints” about mail delivery. For example, if a remote host cannot be contacted, the time of the failure and the suggested next time to try that host are recorded. Any delivery process that has a message for that host reads the hint and skips the delivery if the retry time has not been reached. This does not affect delivery of the same message to other hosts when there is more than one recipient address.

Because the coordinating data is treated as a collection of hints, it is not a major disaster if any or all of it is lost; there may be a period of less optimal mail delivery, but that is all. Consequently, the code that maintains the hints can be simple, because it does not have to be made robust against unusual circumstances.

3.6 How Exim is configured

Configuration information, supplied by the administrator, is used at two different times: one configuration file is used when building the Exim binary, and another is read whenever the binary is run. Most options can be specified in only one of these files. That is, they either control how the binary is built, or they modify its behaviour at runtime. However, there are a few build-time options that set defaults for runtime behaviour. The sources of Exim’s configuration information are shown in figure 3-1.

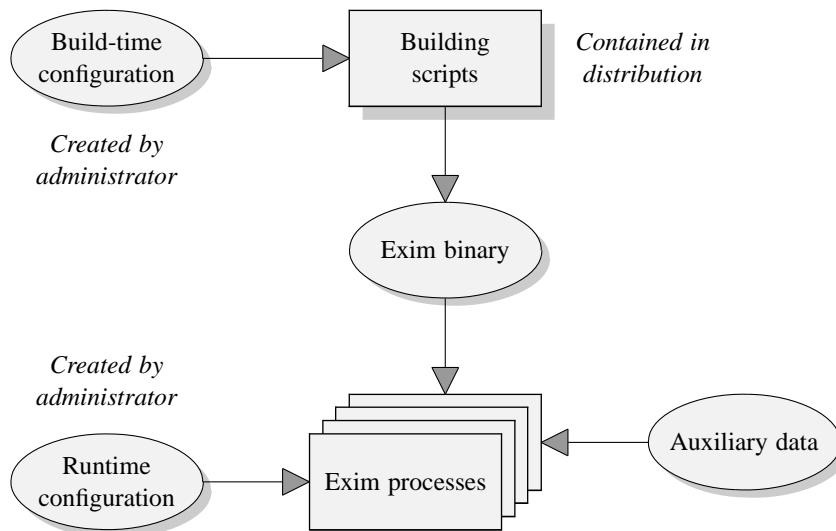


Figure 3-1: Exim configuration

The build-time options are of three kinds:

- Those that specify the inclusion of optional code, for example, to support specific database lookups such as LDAP, or to support IPv6.
- Those that specify fixed values that cannot be changed at runtime, for example, the mode (permissions) of message files in Exim's spool directory.
- Those that specify default values for certain runtime options, for example, the location of Exim's log files.

The process of building Exim from source is described in chapter 25. Here, we consider the runtime configuration. This is controlled by a single text file, often called something like */usr/exim/configure* or */etc/exim.conf*. You can find out the actual name by running the following command:

```
exim -bP configure_file
```

Whenever Exim is executed, it starts by reading its runtime configuration file. A large number of settings can be present, but for any one installation only a few are normally used. The data from the file is held in main memory while an Exim process is running. For this reason, if you change the file, you have to tell the Exim daemon to reload it. This is done by sending the daemon a SIGHUP signal. All other Exim processes are short-lived, so as new ones start up after the change, they pick up the new configuration automatically.

For very simple installations, it may be possible to include all the configuration data within the runtime configuration file. A configuration of this type is shown in the next chapter (☞ 4.3.2). Normally, however, the runtime configuration refers to auxiliary data, which can be in ordinary files, or in databases such as those maintained by MySQL or LDAP. Common examples are the system alias file (usually called */etc/aliases*) and users' *.forward* files. Files or databases can also be used for lists of hosts, domains, or addresses that are to be handled in some special way and that are too long to conveniently include within the configuration file itself. Data from such sources is read afresh every time it is needed, so updates take immediate effect and there is no need to send a SIGHUP signal to the daemon.

The simplest item that is found in the runtime configuration file is an option set to a fixed string. For example, the following line:

```
qualify_domain = example.com
```

specifies that addresses containing only a local part and no domain are to be turned into complete addresses ("qualified") by appending *@example.com*.¹ Each such setting appears on a line by itself. For many option settings, fixed data suffices, but Exim also provides ways for you to supply data that is re-evaluated and modified

¹ Unqualified addresses are accepted only from local processes, or from certain remote hosts that you explicitly designate.

every time it is used. Examples and explanations of this feature are introduced later in this chapter.

3.7 How Exim delivers messages

Exim's configuration determines how it processes addresses; this processing involves finding information about the destinations of a message and how to transport it to those destinations. In this and the following sections, we discuss how the configuration that you set up controls what happens.

There are many different ways an address can be processed. For example, looking up a domain in the DNS involves a completely different way of processing from looking up a local part in an alias file, and delivering a message using SMTP over TCP/IP has very little in common with appending a message to a mailbox file. There are separate blocks of code in Exim for doing the different kinds of processing, and each is separately and independently configurable. The word *driver* is used as the general term for one of these code blocks. In many cases, when you specify that a particular driver is to be used, you need only give one or two parameters for it. However, most drivers have a number of options whose values can be changed to vary the driver's behaviour.

There are three different kinds of driver. Two of them are concerned with handling addresses and delivering messages, and are called *routers* and *transports*. The job of routers is to process addresses and decide what deliveries are to take place. Transports, on the other hand, are the components of Exim that actually deliver messages by writing them to files, or to pipes, or over SMTP connections. The third kind of driver handles SMTP authentication and is described in chapter 13.

Before going into more detail, we take a brief look at the way drivers are used as a message makes its way through the system. Exim has to decide whether each address is to be delivered on the local host or to a remote one. Then it has to choose the correct form of transport for each address (appending to a user's mailbox, for instance, or connecting to another host via SMTP), and finally it has to invoke those transports. For example, in a typical configuration, a message that is addressed to *bug_reports@exim.example*, where *exim.example* is a local domain, might be handled like this:

- (1) The first router in the configuration is a **dnslookup** router that handles addresses that are not in a local domain by looking up MX records in the DNS. As the address we are considering is in a local domain, this router is skipped.
- (2) The next router handles system aliases; this tells Exim to check the */etc/aliases* file. Here it finds that the local part *bug_reports* is indeed an alias, and that it resolves to two other addresses: the local address *brutus@exim.example*, and the remote address *julia@helpersys.org.example*. Exim adds these two new

addresses to the list of recipients that it is routing. It has now finished with the original address.

- (3) The two new addresses are considered in turn. For the first of them, *brutus@exim.example*, the first router is again skipped because the domain is local. This time, the second router does not find an alias for *brutus*, so the address is passed on to the following routers. One of them is a router that recognizes local users such as *brutus*, and it arranges for Exim to run a transport called **appendfile**, which adds a copy of the message to Brutus' mailbox. The actual delivery does not take place until after Exim has worked out how to handle all the addresses.
- (4) For the other recipient, because the domain *helpersys.org.example* is not local, the first router is run. It looks up the domain in the DNS, and finds the IP address of the remote host to which the message should be sent. It then arranges for Exim to run the **smtp** transport to carry out the delivery.

This example has introduced several of the most commonly used drivers. Later in this chapter, we work through a similar example in much more detail. The individual drivers are described in their own sections in later chapters; here is an alphabetical list of them:

accept

A router that accepts any address that is passed to it. Normally, it is constrained by one or more preconditions, as described in the next section. For example, **accept** is used with the `check_local_user` option to accept messages for local users. Without any preconditions, **accept** can be used as a “catchall” router.

appendfile

A transport that writes messages to local files. It can be configured either to append to a single file that holds multiple messages, or to write a new file for each message.

autoreply

A transport that generates automatic replies to messages.

dnslookup

A router that looks up domains in the DNS and does MX processing.

ipliteral

A router that handles “IP literal” addresses such as *user@[192.168.5.6]*. These are relics of the early Internet that are no longer in common use.

lmtp

A transport that delivers messages to external processes using the LMTP protocol (RFC 2033), which is a variation of SMTP that is designed for passing messages between local processes.

manualroute

A router that routes domains using locally supplied information such as a list of domains and corresponding hosts.

pipe

A transport that passes messages to external processes via pipes.

queryprogram

A router that runs an external program in order to route an address.

redirect

A router that handles several different kinds of redirection, including alias files, users' *.forward* files, and Exim or Sieve filters. It can also explicitly force failure of an address.

smtp

A transport that writes messages to other hosts over TCP/IP connections, using either SMTP or LMTP.

The configuration may refer to the same driver code more than once, but with different options, thus allowing for multiple instances of the same driver type. Each driver instance is given an identifying name in the configuration file, for use in logging and for reference from other drivers.

3.8 Processing an address

When routing an address, Exim offers it to each configured router in turn, until one of them is able to deal with it. The order in which routers are defined in the configuration file is therefore important. However, before running a router, Exim first checks a number of preconditions. If any of them are not met, the router is skipped. For example, a router can be configured to apply only to certain domains or local parts. A wide range of conditions can be tested; as an extreme example, you can even restrict routers to certain times of day if you wish. The process of routing an address is illustrated in figure 3-2.

A router that successfully handles an address may assign that address to a particular transport. Alternatively, a router may generate one or more "child" addresses that are added to the message's address list and processed in their own right, with the original

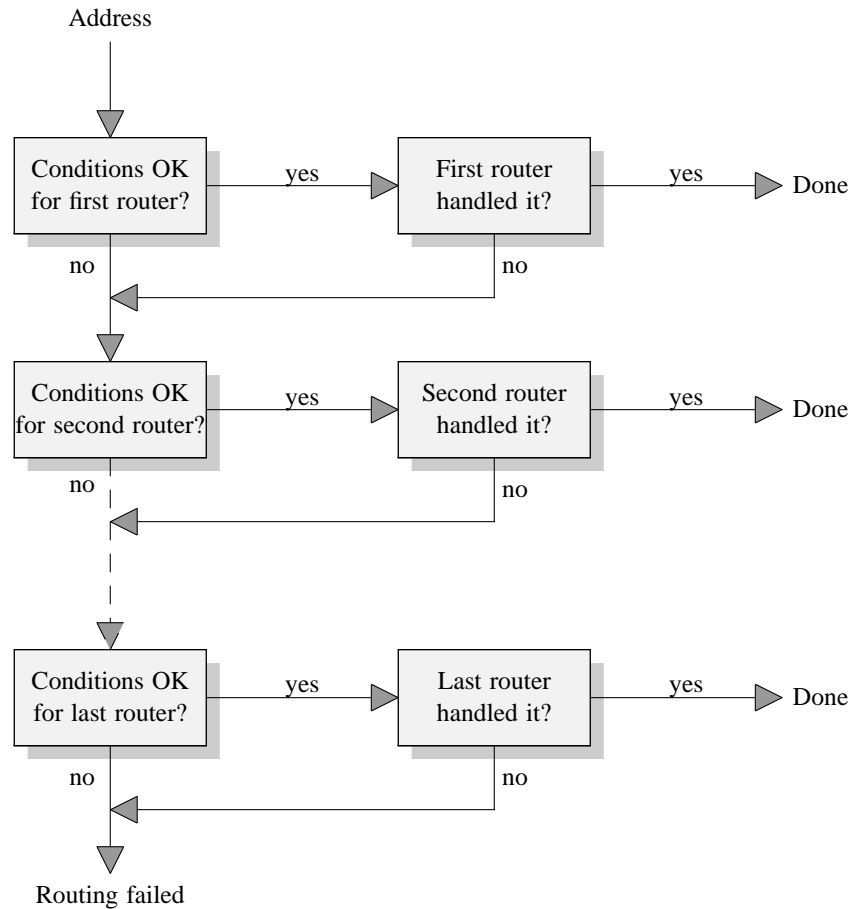


Figure 3-2: Routing an address

address no longer playing any part. This is what happens when a local part matches an entry in an alias list, or when a user's *forward* file is activated.

When a router runs but cannot handle an address, it is said to *decline*. When this happens, the address is by default passed to the next router. If every router declines or is skipped, the address cannot be handled at all, and delivery fails. However, you can cut the routing process short by setting the `more` option on a router to be false. In this case, addresses that cause the router to decline are not passed on. Instead, routing fails immediately. For example, the first router in a configuration is often a **dnslookup** router that has a precondition for checking that the domain is not local, together with the `more` option set false. For a local domain, the router is skipped and the following routers are used, because the precondition is not met. For a non-local domain, the router does a DNS lookup. If it succeeds, the address is assigned to a

suitable transport; if it fails, the router declines, but because of the false setting of `more`, no further routers are run. This means that the subsequent routers can assume that they are dealing with local domains only.

Boolean options such as `more` can be set using the words `true` and `false`, as in this example:

```
more = false
```

An alternative syntax allows the option name on its own for a true setting, and the name preceded by `no_` for a false setting. Thus, it is common to see router configurations containing the line `no_more`, which has the same effect as the example above.

In addition to accepting an address or declining, there are three other returns that a router can give:

defer

The router may be able to handle the address, but it cannot do so at the present time. Typically, this occurs when some data required by the router is inaccessible. For example, a database may be offline, or a DNS lookup may time out. Delivery to the address is postponed until a later delivery attempt.

fail

The router recognizes the address, and knows that routing should fail. For example, you could keep a list of ex-employees and arrange for mail to them to fail with a special message instead of the standard “unknown user”, in an attempt to reduce queries to your postmaster.

pass

The router recognizes the address, but cannot handle it itself. It requests that the address be passed to a later router, overriding a false setting of `more`.

Figure 3-3 illustrates these outcomes for an individual router. However, most routers are restricted to just a few of these possibilities.

3.9 A simple example in detail

To help clarify the mechanisms just described, and to introduce some details of the runtime configuration file, a complete example of a message delivery is presented here. The scenario is a host called *simple.example*, where the host name is the only local mail domain. The host is using a simple Exim configuration file that supports aliases, users’ forward files, delivery to local users’ mailboxes, and remote SMTP delivery. Suppose a user of this host has sent a message addressed to one local and one remote recipient:

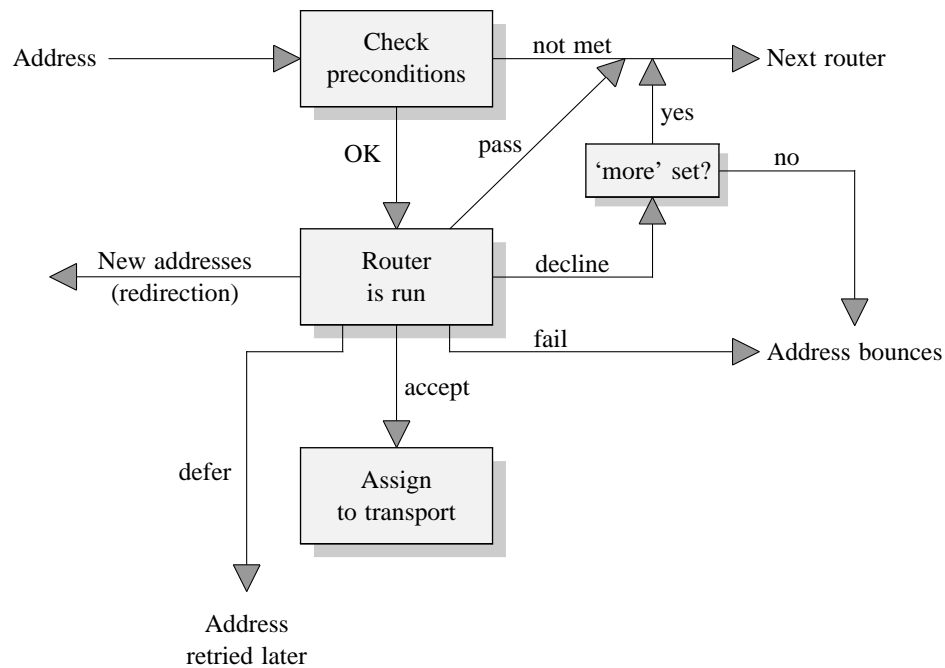


Figure 3-3: An individual router

```

postmaster@simple.example
friend@another.example

```

At the start of delivery, Exim's list of addresses that are to be routed is initialized with the two original recipients, and its first job is to work through this list, deciding what to do for each address. We suppose that it starts by routing *postmaster@simple.example*. The first router in the configuration is this:

```

notlocal:
  driver = dnslookup
  domains = ! simple.example
  transport = remote_smtp
  no_more

```

The first line, terminated by a colon, is the name for this particular router instance, chosen by the system administrator. Each driver instance of a particular type (router or transport) must have a distinct name. The second configuration line specifies which kind of router this is (or, to put it another way, it chooses which block of router code to run), and the remaining lines are options for the router. Options specify both preconditions and parameters that a router uses.

The only precondition on this router is the setting of the `domains` option, which specifies the domains that the router handles. In this case, there is just one domain listed, preceded by an exclamation mark. This is a sign of negation in Exim configuration lists. Thus, this precondition means “the domain must not be *simple.example*”. Consequently, this router is skipped when processing *postmaster@simple.example*, and Exim moves on to the next router, whose configuration is as follows:

```
system_aliases:
  driver = redirect
  data = ${lookup{$local_part}lsearch{/etc/aliases}}
```

The **redirect** router implements address redirection of various kinds, and in this instance it is configured to search the system aliases file (*/etc/aliases*) to see if the local part of the address is an alias.

The value of the `data` option is different from the settings we have met so far, which have all been fixed values. Much of the flexibility of Exim’s configuration comes from the use of option settings where the specified strings contain variables and other special items. Each time such a string is used, it undergoes a process known as *string expansion*, in which the variables are replaced by their current values and the other special items are replaced by the results of processing the text in various ways. String expansion is used in many examples throughout this book, and chapter 20 contains a detailed discussion of this feature. Here, we will just give a brief description of this particular example.

The embedded dollar characters in the string trigger the expansion mechanism. The first of them introduces a lookup item, which is replaced by data that is looked up in a file. The second dollar introduces a variable substitution; in this case, the value of the *\$local_part* variable is being used as the key for the lookup. As you might guess, *\$local_part* contains the local part of the address that is being processed, which in this case is *postmaster*.

The remainder of the lookup item specifies the file, and the way in which it is to be searched. In this case, the file is */etc/aliases*, and a linear search (“*lsearch*”) is required. This expects each line of the file to contain an alias name, optionally terminated by a colon, followed by the list of replacement addresses for the alias, which may be continued onto subsequent lines by starting them with whitespace. A comma is used to separate addresses in the list. For example:

```
root:      postmaster@simple.example,
          herb@simple.example
postmaster: simon@simple.example
```

Notice that the first line specifies that *root* is an alias for *postmaster*, which itself is an alias. This is a common practice, and works exactly as you might expect, though care must be taken to avoid routing loops (☞ 3.10.3). Exim reads through the file,

and finds the entry for *postmaster* with its associated data. The result of the expansion of the `data` option is the string:

```
simon@simple.example
```

The **redirect** router adds this new address to the list of addresses to be routed, and returns a code that indicates success. This means that *postmaster@simple.example* has been completely processed. The list of pending addresses now contains the following:

```
simon@simple.example
friend@another.example
```

Suppose Exim tackles *simon@simple.example* next. This is another local address, so again the **notlocal** router is skipped, and the address is offered to the **system_aliases** router. This time, however, there is no match for *simon* in `/etc/aliases`, so the result of expanding the `data` option is empty. The router returns a code indicating “decline”, which causes Exim to offer the address to the next router, whose configuration is as follows:

```
userforward:
  driver = redirect
  check_local_user
  file = $home/.forward
```

This is another instance of the **redirect** router. This time there is one precondition: the `check_local_user` option. This checks that the local part of the address corresponds to a user login name on the local host. If there is no matching user, the router is skipped. However, we suppose that *simon* is a valid user, so the router is run. The `data` option is not set for this instance of **redirect**; instead there is a setting of the `file` option. The value of the option is expanded (with `$home` replaced by *simon*’s home directory), but instead of being a list of redirection addresses (as `data` was for the previous router), the expanded string is interpreted as a file name. The router now checks to see if the file exists.

If *simon* has a `.forward` file in his home directory, its contents are a list of forwarding addresses and other types of item (⇨ 7.6.7). The addresses are added to the list of addresses to be routed, and the **userforward** router returns a code indicating success. The new addresses are eventually processed independently, in the same way that the new address from the **system_aliases** router was handled.

If *simon* does not have a `.forward` file, the router declines, and *simon@simple.example* is offered to the next router in the configuration:

```
localuser:
  driver = accept
  check_local_user
  transport = local_delivery
```

This has the same precondition (`check_local_user`) as the previous router. Exim keeps a cache of the most recently looked-up username to avoid wasteful repetition, so it already knows that this precondition is met. If *simon* were not a local user, the router would decline, and as there are no more routers in this configuration, the address would fail. It would be placed on a list of failed addresses and used to generate a bounce message at the end of the delivery attempt.

The **accept** router imposes no internal conditions of its own. If its preconditions are met, it accepts the address and assigns it to the transport that is specified by the `transport` option (in this case, **local_delivery**). The uid, gid, and home directory that were looked up for *simon* are attached to the address, so that they can be used by the transport.

That is all that happens at this stage; no actual delivery takes place until later. The processing of *postmaster@simple.example* is illustrated in figure 3-4, where the ellipses represent sources of information, and the rectangles represent routers. The **notlocal** router, which is skipped for this address, is not shown.

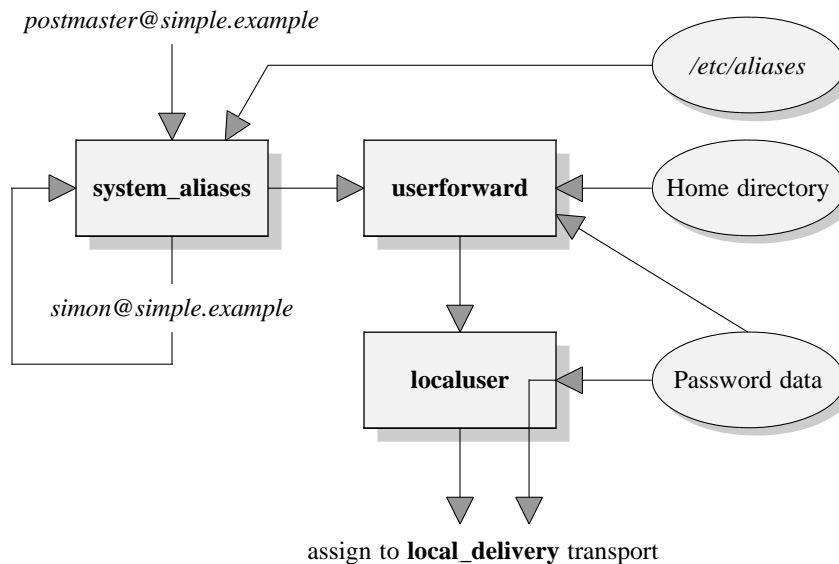


Figure 3-4: Routing *postmaster@simple.example*

There is still one address to process: *friend@another.example*. Exim offers it to the first router:

```
notlocal:
  driver = dnslookup
  domains = ! simple.example
  transport = remote_smtp
  no_more
```

This time, the precondition is met because the domain is not *simple.example*, and so the router is run. The job of **dnslookup** is to obtain a list of remote hosts by looking up the domain in the DNS, using MX and address records (↔ 2.8). If it fails to find the domain in the DNS, the router declines. In this configuration, we have the `more` option set false (by virtue of the `no_more` line), which means that no further routers are run. In other words, if the domain is not found in the DNS, the address is bounced.

When **dnslookup** is successful, it ends up with an ordered list of hosts and their IP addresses. It assigns the email address to the **remote_smtp** transport, attaching the host list to the address. In our example, if the MX and address records were the following:

```
another.example.      MX  6  mail-2.another.example.
another.example.      MX  4  mail-1.another.example.
mail-1.another.example. A    192.168.34.67
mail-2.another.example. A    192.168.88.32
```

then the list of hosts to be passed with the address to **remote_smtp** would be:

```
mail-1.another.example 192.168.34.67
mail-2.another.example 192.168.88.32
```

Any hosts that have the same MX preference value are sorted into a random order. The processing of *friend@another.example* is illustrated in figure 3-5.

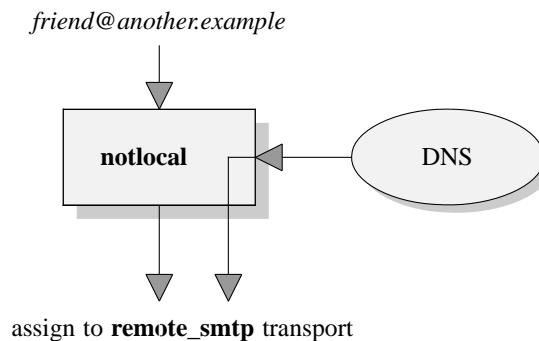


Figure 3-5: Routing *friend@another.example*

There are now no more unprocessed addresses, so the routing phase of the delivery process is complete. Exim moves on to carry out the actual deliveries by running the

transports that have had addresses assigned to them. Local transports are run first; in our example, one local delivery is set up for the address *simon@simple.example*, using the **local_delivery** transport. This was specified by the **localuser** router that handled the address. The transport is configured thus:

```
local_delivery:
  driver = appendfile
  file = /var/mail/$local_part
  delivery_date_add
  envelope_to_add
  return_path_add
```

This specifies an **appendfile** transport (☞ 9.4), which adds a copy of the message to the end of a mailbox file in conventional Unix format when configured in this way. The name of the file is given by the `file` option, which is an expanded string. Exim replaces the substring `$local_part` by the local part of the address that is being delivered, so the file that is actually used is */var/mail/simon*. The remaining three options request the addition of three generally useful header lines as the message is written:

Delivery-date:

A header line that records the date and time of delivery, for example:

```
Delivery-date: Fri, 31 Dec 1999 23:59:59 +0000
```

Envelope-to:

A header line that records the original recipient address (the “envelope to” address) that caused the delivery; in this example it would be:

```
Envelope-to: postmaster@simple.example
```

Preserving this address is useful in case it does not appear in the *To:* or *Cc:* header lines.

Return-path:

A header line that records the sender from the message’s envelope. For example:

```
Return-path: <user@simple.example>
```

For bounce messages, which have no sender, it looks like this:

```
Return-path: <>
```

Local deliveries are run sequentially in separate processes that change their user identity to an appropriate value in each case (☞ 22.1.1). In this case, the user ID (uid) and group ID (gid) of the local user were passed to the transport by the **localuser** router, so these are used. The delivery subprocess is therefore running “as the user” when it accesses the mailbox.

When the subprocess has finished, there are no more local deliveries, so Exim proceeds to the remote ones. These are also run in separate processes, in this case using Exim's own uid and gid. Exim can be configured to run a number of remote deliveries in parallel when there is more than one remote delivery to be done, but in our example there is only one remote delivery. This was set up for *friend@another.example* by the **notlocal** router, which assigned it to the **remote_smtp** transport:

```
remote_smtp:  
  driver = smtp
```

There are no option settings here beyond the one that selects the type of transport, because the list of hosts was obtained by the **notlocal** router and passed to the transport along with the address. The parameters of the outgoing SMTP connection (for example, the timeouts) can be changed by other options, but in this case we accept all the defaults. The **smtp** transport tries to make an SMTP connection to each host in turn. If all goes well, a connection is made to one of them, and the message is transferred.

There are now no more deliveries to be done, and all the recipients have been successfully handled, so at this point Exim can delete the message files on its spool and log the fact that this message has been delivered. The delivery process then exits.

3.10 Complications while routing

Things do not always go as smoothly as described in the simple example. Some of the more common complications that can be encountered when routing an address are given below.

3.10.1 Duplicate addresses

Duplicate addresses are a complication that Exim may have to handle, either because the sender of the message specified the same address more than once, or because aliasing or forwarding duplicated an existing recipient address. For any given address, only a single delivery takes place, except when the duplicates are pipe commands. If one user is forwarding to another, and a message is sent to both of them, only a single copy is delivered. If, on the other hand, two different users set up their *.forward* files to pipe to */usr/bin/vacation* (for example), a message that is sent to both of them runs the vacation program twice, once as each user.

3.10.2 Missing data

Sometimes, a router is unable to determine whether it can handle an address. For example, if the administrator has misspelled the name of an alias file, or if it has been accidentally deleted, a **redirect** router cannot operate. Timeouts can occur when

a router queries the DNS, and routers can refer to databases that may at times be offline. In these situations, the router returns a code indicating “defer” to the main part of Exim, and the address is neither delivered nor failed, but left on the spool for another delivery attempt at a later time. The control of retry times is described in chapter 12. If the error condition is felt to be sufficiently serious, the message is “frozen”, which means that queue runner processes will not try to deliver it. Because frozen messages are highlighted in queue listings, this also serves to bring them to the administrator’s attention.

3.10.3 Routing loops

When a **redirect** router handles an address, the new addresses that it generates are each processed afresh, just like the original recipient addresses.¹ This means that one alias can refer to another, as in the example we showed earlier:

```
root:      postmaster@simple.example
postmaster: simon@simple.example
```

However, this opens up the possibility of routing loops. To prevent this, Exim automatically skips a router if the address it is handling has a “parent” address that was processed by that router. Consider the following broken alias file:

```
chicken:   egg@simple.example
egg:       chicken@simple.example
```

This router turns the address *chicken@simple.example* into *egg@simple.example*, and then turns it back into *chicken@simple.example* the next time through. However, on the third pass, Exim notices that the address was previously processed by the router, so it is skipped and the next router is called. The chances are that the resulting delivery or bounce are not what was intended, but at least the loop is broken.

3.10.4 Remote address routing to the local host

After Exim has routed an address to a list of remote hosts, it checks to see whether the first host on the list is the local host. Usually, this indicates some kind of configuration error, and by default Exim treats it as such. However, there are types of configuration where this is legitimate, and for these cases, the `self` option can be used to specify what is to be done (☞ 6.6).

3.11 Complications during delivery

A successful routing process for a remote address discovers a list of hosts to which it can be sent, but it cannot check the local part of the address. The most common permanent error during a remote delivery is “unknown user” in response to an SMTP

¹ This is the normal practice; there are occasions when it is not wanted, and there is an option, `redirect_router`, that can be used to disable it.

RCPT command. Responsibility for the message remains with the sending host, which must return a bounce message to the sender.

There are other reasons a remote host might permanently refuse a message, and in addition, there are many common temporary errors, such as the inability to contact a host. These cause a message to remain on the spool for later delivery.

In contrast, routers for local addresses normally check local parts, so any “unknown user” errors happen at routing time. The only problems a local transport is likely to encounter are errors in the actual copying of the message. The most common is a full mailbox; Exim respects system quotas and can also be configured to impose its own quotas (☞ 9.4.7). A quota failure leaves the message on the spool for later delivery.

The runtime configuration contains a set of retry rules (☞ 12.5) that specify how often, and for how long, Exim is to go on trying to deliver messages that are suffering temporary failures. The rules can specify different behaviours for different kinds of error. For example, a retry rule can specify that there are no retries for quota failures.

3.12 Complications after delivery

When all delivery attempts for a message are complete, a delivery process has two final tasks. If any deliveries suffered temporary errors, or if any deliveries succeeded after previous temporary errors, the delivery process has to update the retry hints database. This work is saved up for the end of delivery so that the process opens the hints database for updating only once at most, and for as short a time as possible. If the updating should fail, the new hint information is lost, but previous hint information remains. In practice, except in exceptional circumstances such as a power loss, hint information is rarely lost.

Finally, unsuccessful delivery may cause a message to be sent to the sender. If any addresses failed, a single bounce message is generated, containing information about all of them. If any addresses were deferred, and have been delayed for more than a certain time, a warning message may be sent (☞ 22.9.3). Exim sends such messages by calling itself in a subprocess. Failure to create a bounce message causes Exim to write to its panic log and immediately exit. This has the effect of leaving the message on the spool so that there will be another delivery attempt, and presumably another attempt at sending the bounce message when the delivery fails again. Failure to create a warning message, on the other hand, is not treated as serious. Another attempt to send it is made when the original message is processed again.

3.13 Use of transports by routers

In the simple example we have been considering, the **localuser** and **notlocal** routers both include the `transport` option (referring to the **local_delivery** and **remote_smtp** transports, respectively), whereas the other routers do not have any transport settings. A transport is required for any router that actually sets up a message delivery, in order to determine how the delivery should be carried out. When a router is just changing the delivery address by aliasing or forwarding, a transport is not required because no delivery is being set up at that stage.

The **redirect** router has additional options for some special-purpose transports. This router can deliver a message to a specific file, or to a pipe associated with a given command.

3.13.1 Redirecting to a pipe

A line in an alias file of the form:

```
majordomo: |/usr/mail/majordomo <options>
```

specifies that a message addressed to the local part *majordomo* is to be passed via a pipe to a process running the command:

```
/usr/mail/majordomo <options>
```

The other entries in the alias file may just be changing delivery addresses, and may therefore not require a transport. However, this line is setting up a delivery, and so a transport is required. We can add the following line to the **system_aliases** router configuration:

```
pipe_transport = alias_pipe
```

This tells Exim which transport to run when a pipe is specified in the alias file. The transport itself is very simple:

```
alias_pipe:  
  transport = pipe  
  ignore_status  
  return_output
```

A **pipe** transport runs a given command in a new process, and passes the message to it using a pipe for its standard input. In this example, the command is provided by the alias file, so the transport does not need to define it.¹ Setting `ignore_status` tells Exim to ignore the status returned by the command; without this, any value other than zero is treated as an error, causing the delivery to fail and a bounce message to be returned to the sender.

¹ When a **pipe** transport is referenced from the `transport` option of a router other than **redirect**, the command to be run is defined using the transport's `command` option.

Setting `return_output` changes what happens if the command produces output on its standard output or standard error streams. By default, such output is discarded, but if `return_output` is set, the production of such output is treated as an error, and the output itself is returned to the sender in the bounce message.

There is one piece of information that the **alias_pipe** transport needs that we have not yet given, and that is the uid and gid under which it should run the command. When a pipe is triggered by an entry in a user's *forward* file, the user's identity is assumed by default, but when an alias file is used, as it is here, there is no default. The `user` (and, optionally, `group`) option can appear in either the router or the transport's configuration, so the transport could become:¹

```
alias_pipe:
  transport = pipe
  ignore_status
  return_output
  user = majordom
```

3.13.2 Redirecting to a file

In addition to delivery to pipes, aliasing and forwarding may also specify files into which messages are to be delivered. For example, if user *caesar* has a *forward* containing:

```
caesar@another.domain.example, /home/caesar/mail-archive
```

it requests delivery to another mail address, and also into the named file, which is a delivery that needs a transport. To support this feature, the **userforward** router could contain:

```
file_transport = address_file
```

This tells Exim which transport to run when a filename is specified instead of an address in a forward file. The transport itself is very simple:

```
address_file:
  driver = appendfile
```

The filename comes from the forward file, and all other options are defaulted.

An alias or forward file may contain both these kinds of entry, thus requiring both `pipe_transport` and `file_transport` to be given on a single router. These options are used for these very specific purposes only, and should not be confused with the generic `transport` option that can be set for any router instance.

¹ This assumes that all the pipes specified in the alias file are to be run under the same uid. If there are several instances that require different user identities, an expansion string can be used to select the correct uid, but that is too advanced for the discussion here.