Some thoughts on MTA architecture

`http://dotat.at/writing/mta-arch`

Tony Finch
⟨`fanf2@cam.ac.uk`⟩ ⟨`dot@dotat.at`⟩

Monday 2 June 2008

Abstract

The currenly popular MTAs were all designed before the spam problem became severe. Because of this some of their basic architectural decisions are unhelpful in the face of current email loads. What would a 21st century spam-conscious MTA design look like? This talk discusses some possible approaches.

1 Introduction

About me

My CV expressed as a list of email addresses:

1994 – 1997	⟨`fanf2@cam.ac.uk`⟩	computer science
1997 – 2000	⟨`fanf@demon.net`⟩	web server admin
2000 – 2001	⟨`fanf@covalent.net`⟩	Apache httpd coder
2002 – now	⟨`fanf2@cam.ac.uk`⟩	postmaster
1997 . . .	⟨`dot@dotat.at`⟩	
1999 . . .	⟨`fanf@apache.org`⟩	httpd
2002 . . .	⟨`fanf@FreeBSD.org`⟩	unifdef
2004 . . .	⟨`fanf@exim.org`⟩	
2006 . . .	⟨`fanf@apache.org`⟩	SpamAssassin

Tony Finch is a Unix system developer. He has contributed to a number of open source projects, including Exim, FreeBSD, and the Apache httpd. He works for the University of Cambridge Computing Service where he runs the university's central SMTP relay.

“Wouldn’t it be nice if. . . ?”

- theoretical musings on MTA architecture
- originally a series of postings on my blog, Feb 2006 – March 2007
- there is no code and no likelihood of code

I need to emphasize up-front that this is *not* an implementation report!

The original posts on my LJ were titled “How not to design an MTA”. This talk aims to be more positive.

The most popular MTAs were designed 10 or more years ago, before the spam problem became overwhelmingly serious.

A snapshot of the problem

Average email traffic (legitimate and spam):

Mar 2005	15
Mar 2006	20
Mar 2007	35
Mar 2008	80

- all numbers in messages (or rejections) per second

Current traffic classification:

relay attempts	0.5 – 1.5
known malware	2 – 4
blacklisted	60 – 75
invalid recipient	1.5
invalid sender	1.2
SpamAssassin	2
legitimate email	3
internal email	2.5

These numbers come from the University of Cambridge’s central email relay which has about 30,000 users. The point is to notice which numbers are bigger and therefore understand where we need to concentrate effort on performance.

Note for example that the total number of address verifications for rejected messages is about 5/sec which is nearly as much as the volume of desirable email.

In this table, “legitimate email” covers email from outside the University and does not include “internal email”, so the volume of desirable email is about 5.5 per second. “Known malware” is identified from HELO hostname signatures.

Legitimate external email is about 4% of the total, and internal email is about 3% of the total. About 86% is rejected before address verification.

2 Concurrency

Concurrency

- concurrency requirements grow with spam volumes
- most MTAs use an OS process per connection
- really inefficient!

Let's start off with the uncontroversial observation that most MTAs still have really old-school inefficient concurrency architectures. This is aggravated by some anti-spam techniques that slow down SMTP conversations, such as "greet-pause" (the server delays its greeting in order to spot abusive clients that don't wait their turn to speak) and "teergrubing" (an attempt to waste spammers' resources by delaying all SMTP responses — which is fairly pointless when the spammers' botnets have more resources than the good guys).

Waste vs efficiency

- event-driven connection multiplexing
- high-level languages with lightweight threads

better software performance $\Longrightarrow$ better hardware efficiency

Instead of throwing hardware at the problem because it maxes out at about 1000 concurrent connections, use `epoll` or `kqueue` to multiplex 10,000+ connections[1]. Even better, write the program in a concurrency-oriented language that has the infrastructure built-in, such as Erlang.

[1] see `http://www.kegel.com/c10k.html`

Waste vs efficiency

- best use of the available resources ...

Bikes are perhaps a better analogy to lightweight concurrency than comparing a Hummer with a Prius.

Some partial solutions

- `SAUCE` – software against UCE `http://www.chiark.greenend.org.uk/~ian/sauce/` (written in Tcl)
- `qpsmtpd-async` – anti-spam smtpd for qmail `http://smtpd.develooper.com/` (written in Perl)
- MailChannels Traffic Control$^{\text{TM}}$ `http://www.mailchannels.com/products/traffic-control.html`

(It so happens that these examples support my recommendation to write in a higher-level language if you want better concurrency.)

These are "partial solutions" because they only deal with incoming connections. SAUCE and MailChannels work as SMTP proxies in front of an existing SMTP server, whereas qpsmtpd uses lower-level (non-SMTP) ways to inject the message into the back-end MTA queue.

They do not address poor concurrency architecture in message routeing and delivery.

3 Verification

Address verification

- most verifications are for messages that will be rejected
- email address routeing can be arbitrarily complicated so verification can be too!
- concurrency useful for multi-recipient messages as well as multiple messages

Address verification is the first difficult anti-spam check that an MTA must perform — DNS blacklists and protocol checks are relatively cheap.

I'll expand on these points in the next few slides, but first a bit about the importance of thorough verification.

Avoid bouncing

- reject unwanted email as early as possible
- try hard not to accept and bounce
- reduce spam backscatter & forwarded spam
- avoid wasting your MTA's resources

If you accept email and later discover you can't deliver it, RFC 2821 says you are supposed to send a bounce message. But what if it's undeliverable because it's spam? The sender address is forged and you'll be sending an unwanted bounce to an innocent third party. You don't want to drop the message in case of a legitimate misconfiguration.

So you must reject incorrectly-addressed email, instead of accepting it then bouncing it. Spam senders drop messages that are rejected. Legitimate senders will generate their own delivery failure reports.

3.1 Routeing is verification

How email addresses are routed

- DNS — MX/A/AAAA
- flat files — text or cdb
    - `aliases`
    - `mailertable`
    - `virtusertable`
- LDAP — "laser" schema
- SQL databases

A lot of the complexity in MTAs exists to support the variety of ways that postmasters want to configure the redirection and routeing of email. I've made particular mention of sendmail's routeing and redirection tables because they are typical of the baroque lookup semantics that MTAs support. In the virtusertable you can match all or part of an address, or you can do wildcard matching, and the destination can be another address or a custom error message …

A few notes on performance. DNS lookups can be very high latency but don't individually require many resources. An MTA's routeing engine should support highly concurrent DNS resolution — not one process per message. On the other hand, LDAP and SQL connections are relatively heavyweight, so an MTA should be able to limit the number of concurrent database connections and schedule lookups across the connection pool.

There are a couple of common email routeing features that make verification particularly tricky.

User-defined filtering

- Sieve — RFC 5228
- address validity can be conditional on the sender's address
- selective sub-address validity, e.g. `fanf9+subaddress@hermes.cam.ac.uk`

First, a practical example. User-defined filtering means that email address validity can change very dynamically, the moreso the larger your userbase is.

Routeing with regular expressions

- try to match address against a series of regular expressions
- when one matches, replace address with corresponding result
- interpolate captured subexpressions
- route resulting address, repeating `regsub` if necessary

Second, one for the computer scientists. It's well known that Sendmail's rewriting language is Turing-complete. I showed in 2002 that Exim is also Turing-complete by writing a configuration that does SK combinator reduction. But any MTA that supports iterated regular expression match and substitution is Turing-complete. This includes Postfix — which implies that its `trivial-rewrite(8)` daemon is in fact non-trivial.

Verification: you're doing it wrong!

Postfix `local_recipients_map`

Postfix and qmail are designed to have heavy separation between their SMTP front-end and their address routeing code. This means that they cannot conveniently use the routeing engine to verify addresses. Consequently, qmail doesn't bother, and Postfix has the `local_recipients_map` option to configure verification performed by the SMTP front-end independently of the routeing engine. This violates the Don't Repeat Yourself ("DRY") principle and is instead more like Write Everything Twice ("WET").

Features like user-defined filters mean that a table defined by the postmaster can only be an approximate list of valid addresses, which means some undeliverable email will accepted then bounced.

3.2 Callout verification

Verifying relayed addresses

We have a number of departments who run their own email servers, but we provide the MX service. When we route email to them we only look at the domain part of the email address, which determines the destination host. This means we don't know which local parts (usernames) are valid and which are not. How can we avoid the accept-and-bounce problem in this situation?

Verification: you're doing it wrong!

- copy table of valid recipients from department to MX
- configure MX to query department's LDAP directory

A couple of ways you might solve the problem boil down to re-implementing Postfix's `local_recipients_map`, but with greater separation between the front and back parts of the system. You have the same problems of over-estimating validity and writing everything twice.

Call-forward recipient verification

```
220 mx.cam.ac.uk
HELO dotat.at
250 Hello
MAIL FROM:<dot@dotat.at>
250 OK
RCPT TO:<?@cl.cam.ac.uk>  220 mta.cl.cam.ac.uk
                          HELO mx.cam.ac.uk
                          250 Hello
                          MAIL FROM:<dot@dotat.at>
                          250 OK
                          RCPT TO:<?@cl.cam.ac.uk>
                          550 Unknown user
550 Unknown user          QUIT
RSET                      221 Goodbye
...
```

A slightly abbreviated example of how call-forward verification works. I am sending email to an address at a department that runs its own mail server. When my SMTP sender states the recipient address, the MX routes the address to verify it, and finds that it is at a remote server. It connects to that server and starts an SMTP conversation which it will abort after getting the recipient verification result. It can then pass the result back to my SMTP sender.

Advantages: No special arrangements between the MX and the departments: if the MX can deliver email to the department, and if the department's server is properly configured not to accept-then-bounce, then the MX can verify addresses at the department's domain. No duplicated configuration.

Caveats: The MX needs to maintain a cache of verification results, so that if it is being hammered it doesn't pass all of its load to the back end.

The MX also needs a way of dealing with callouts that take longer than one SMTP timeout period. Ideally it will either tell the client to come back later, or accept the message and hope for the best. In either case it should continue the callout operation so that it can put a definitive result in its cache. This should be easy to do if the callout uses the MTA's normal delivery mechanisms, which are naturally decoupled from the front end.

The MTA should be designed to make callouts as efficient as possible, since most of them are going to be for messages that are rejected. In particular it must not require disk transactions like Postfix's implementation does.

Verification should be built-in to the MTA. The `milter-ahead` addon to Sendmail has to re-implement Sendmail's routing features — `virtusertable` and `mailertable` in order to implement call-forward verification. Yet another instance of "write everything twice".

4 Content scanning

Content scanning

- anti-spam
- anti-phishing
- anti-virus
- lots of CPU
- lots of memory

Programs like SpamAssassin and ClamAV.

Content scanning goals

- decouple scanner from client concurrency & speed
- do not require entire message to be buffered in RAM
- avoid temporary on-disk buffers
- security boundary between content scanner(s) and MTA

The first requirement means that we should buffer the message while it is being received — for example, while it is being dribbled up a slow DSL line from a compromised home computer. Once the entire message has been received then we can run the scanner(s) over it as fast as they can go. This minimizes the number of concurrent scanners we need to run for a given number of messages per second.

Aside: The milter interface is designed to dribble the message from the MTA to the scanner in blocks as it is received.

The second requirement means that we should write the message to a file on disk, and rely on the operating system's buffer cache to keep the message in RAM if there is space, or page it out if there is not.

The third requirement means that the file we use should be the message's final resting place in the MTA's queue — there must be no unnecesasry copying of the data or filesystem operations (such as rename).

Content scanners are doing a difficult job with untrusted data so ideally they could be run in a separate process with a different user ID than the MTA – which is in fact a fairly natural setup.

Data callout

- use the normal local delivery mechanism
- efficiently transfer a file from the queue to a program
- cross security boundaries
- control concurrency and smooth load spikes

Most of the goals are naturally fulfilled if the MTA has a "data callout" mechanism. Like verification callouts, the front end asks the back end to perform a task for it during the SMTP conversation. Data callouts include the message data, instead of stopping after the envelope.

There are some other interesting things that this mechanism makes much easier. You can use a remote data callout to implement application-level replication, so that the message is stored redundantly in two local MTA queues before the sender gets confirmation that it has been accepted. Or you can implement early delivery, in which you see if the message can be delivered before returning confirmation to the sender, so it only has to be `fsync`ed into the queue if the delivery can't be completed within a short timeout.

5 Log-structured queue

Queue layout

- MTAs typicall scatter messages all over the disk
- often separate files for envelopes and contents
- this makes queue runs particularly expensive

Two files per message (data and metadata) is twice as expensive as it needs to be. Even one file per message (combined envelope and contents) is too expensive: How do you find which message needs to be retried next? Your disks are doing lots of seeks for random-access scanning of the queue.

Log-structured queue

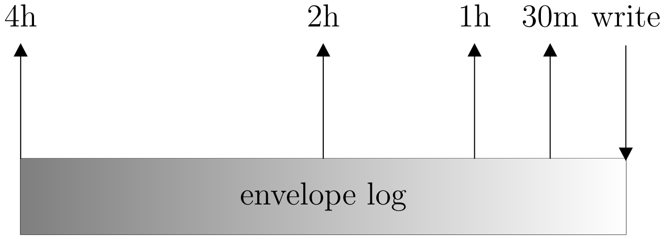
- write all metadata sequentially to one file
- queue runners read file sequentially
- updated envelopes also appended to the file
- queue runners act as garbage collectors
- size of log bounded by retry interval

The idea here is that message contents still written to one file per message, so that we can let the kernel allocate space for it. We only need to look at that file during delivery. Queue running and routeing only need to look at the message envelope, and the log structure makes it really easy to find which one to examine next because the order it is written is the order in which messages need to be retried.

When a message is delivered to a recipient you need to update its envelope to delete that recipient, and when there are none left mark the envelope completed. Log-structured files are append-only, so when you complete a delivery attempt you by write a replacement envelope to the end of the file. The original envelope then becomes garbage.

This has rather beautiful consequences.

Log-structured queue



Most new envelopes written to the log immediately become garbage, because most deliveries succeed first time. Replacement envelopes have a retry time which indicates which of the log readers will deal with it when they get to that point. When a log reader handles an envelope it becomes garbage. The queue runner that processes the oldest records in the log ensures that it leaves no live records behind. This naturally bounds the size of the queue.

The traditional problem with log-structured filesystems and databases is the cost of garbage-collecting old log records — that is, identifying which ones are still live. This isn't a problem for an MTA queue because the queue runners need to scan old log records anyway, so you get garbage collection for free.

The log structure also allows you to reduce `fsync` operations. Deliveries must be synced as quickly as possible; however when you are accepting a message then you can delay sending the confirmation to the client in the hope that a delivery sync will do your job for you.

It probably also makes sense to put small messages in the log, to save the cost of creating, syncing, and deleting a message contents file.

Aside: A fully-engineered log will probably be divided a file for the active head, and another that contains any envelopes that were not completely delivered first time. A really busy machine might have multiple head files on different disks.

6 Conclusion

Architectural principles

- lightweight concurrency througout the system
- load smoothing / scheduling of scarce resources
  - database connections, content scanners
- address routeing is verification
- content scanning is a data call-forward
- a log-structured queue minimizes disk seeks

The same mechanism in the MTA is used to implement routeing and verification, including all the concurrency and resource management. The delivery mechanisms are used to implement call-forward recipient verification and content scanning.

That's all, folks!

- slides and notes available online: `http://dotat.at/writing/mta-arch`
- any questions?