Parsing Log Files Produced by the Postfix Mail Transfer Agent

John Tobin

A thesis submitted to the University of Dublin, in fulfilment of the requirements for the degree of Master of Science in Computer Science.

April 2009

Declaration

I declare that this thesis, and the work described herein, is entirely my own work, and has not been submitted as an exercise for a degree at this or any other university. This thesis may be borrowed or copied upon request with the permission of the Librarian, Trinity College, University of Dublin.

John Tobin
October 9, 2009

Acknowledgements

I am indebted to my supervisor, Dr. Carl Vogel, for his advice, assistance, and guidance.

I am grateful to my wife, Ariane Tobin, for her support, patience, and encouragement; without her, I would not have accomplished this.

Abstract

Postfix is a widely used Mail Transfer Agent, transferring hundreds of millions of mails between senders and recipients every day. Parsing log files produced by Postfix is much more difficult than it first appears, but it *is* possible to achieve a high degree of accuracy in understanding Postfix log files, and subsequently in reconstructing the actions taken by Postfix when processing mail delivery attempts. This thesis describes the creation of a parser for Postfix log files, documenting the architecture developed for this project and the parser that implements it, the difficulties encountered and the solutions developed. The parser stores data gleaned from the log files in an SQL database; future projects or programs could use the gathered data to optimise current anti-spam measures, to produce statistics showing how effective those measures are, or to provide a baseline to test new anti-spam measures against. The Postfix log file parser needs to be very precise and strict when parsing, yet must allow users to easily adapt or extend it to parse new log lines, without requiring that the user have an in-depth knowledge and understanding of the parser's internal workings. The newly developed architecture is designed to make the process of parsing new inputs as simple as possible, enabling users to trivially add new rules (to parse variants of known inputs) and relatively easily add new actions (to process a previously unknown category of inputs). The parser implemented for this project is evaluated on the criteria of efficiency and coverage of Postfix log files, demonstrating that the conflicting goals of efficiency and accuracy can be balanced, and that one need not be sacrificed to achieve the other.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The architecture and implementation described in this thesis were developed as the foundation of a larger project to improve anti-spam defences, by analysing the performance of the set of anti-spam techniques currently in use, optimising the order and membership of the set based on that analysis, and developing supplemental anti-spam techniques where deficiencies are identified. Most anti-spam techniques are content-based (e.g. [2, 23, 31]) and require a mail to be accepted before determining if it is spam, but rejecting mail during the delivery attempt is preferable: senders of non-spam mail that is mistakenly rejected will receive an immediate non-delivery notice; resource usage is reduced on the accepting mail server, allowing more intensive content-based techniques to be used on the remaining mail that is accepted; users have less spam mail to wade through. Improving the performance of anti-spam techniques that are applied when mail is being transferred via Simple Mail Transfer Protocol (SMTP) is the goal of this project, by providing a platform for reasoning about the performance of anti-spam techniques.

The approach chosen to measure performance is to analyse the log files produced by Postfix,[1] the Mail Transfer Agent (MTA) used by the School of Computer Science and Statistics, rather than modifying its source code to generate statistics: this approach improves the chances of other Postfix users testing, using, and benefiting from the software developed for this project.

[1] `http://www.postfix.org/` (last checked 2009/04/21). An introduction to Postfix is provided in §2.3 [p. 22].

The need arose for a parser capable of dealing with the great number and variety of log lines produced by Postfix: the parser must be designed and implemented so that adding support for parsing new inputs is a simple task, because the log lines to be parsed will change over time. This variety in log lines occurs for several reasons:

- Log lines differ amongst versions of Postfix.
- The mail server administrator can define custom rejection messages.
- Policy servers (see §2.3.2 [p. 26]) may log different messages depending on the characteristics of the connection.
- Every DNS Blacklist (DNSBL)[2] returns a different explanatory message.

Most mail server administrators will have performed some basic processing of the log files produced by their mail server at one time or another, whether it was to debug a problem, explain to a user why their mail is being rejected, or check if new anti-spam techniques are working. The more adventurous will have generated statistics to show how successful each of their anti-spam measures has been in the last week, and possibly even generated some graphs to clearly illustrate these statistics to management or users.[3] Very few will have performed in-depth parsing and analysis of their log files, where the parsing must correlate each connection or mail's log lines rather than processing each log line in isolation. One of the barriers to this kind of processing is the unstructured nature of Postfix log files, where each log line was added on an ad hoc basis when a requirement was discovered or new functionality was added.[4] Further complication arises because the set of log lines is not fixed, and log lines can differ in many ways between servers, even within the same organisation, where servers may be configured differently or running different versions of Postfix.

[2]This thesis is supplied with a glossary (§B) and a list of acronyms (§C).

[3]This was the first real foray the author, a Systems Administrator for a network of over 2000 computers and over 1800 users, took into processing Postfix log files.

[4]A history of all changes made to Postfix is distributed with the source code, available from `http://www.postfix.org/` (last checked 2009/02/23).

The only prior published work on the subject of parsing Postfix log files that the author is aware of is *Log Mail Analyser: Architecture and Practical Utilizations* [11], which aims to extract data from Postfix log files, correlate it, and present it in a form suitable for a systems administrator to search using the myriad of standard Unix text processing utilities already available; it is reviewed alongside the other parsers in the State of the Art Review in chapter 3. It was hoped to reuse an existing parser rather than writing one from scratch, but the effort required to adapt and improve an existing parser was judged to be greater than the effort to write a new one, as described in the State of the Art Review.

Once it was decided that a new parser would be written, an architecture was required to base the implementation on. Existing architectures are tailored towards parsing inputs with a fixed grammar or a tightly constrained format, whereas Postfix log files lack any form of constraint, as outlined earlier. A new architecture was designed and developed for this parser, with the hope that it will be useful to others. The resulting architecture is conceptually simple: provide a few generic functions (*actions*), each capable of dealing with an entire category of inputs (e.g. rejecting a mail delivery attempt), accompanied by a multitude of precise patterns (*rules*), each of which recognises one input variant and only that variant (e.g. rejection by a specific DNSBL), and specifies which action will process the inputs it recognises. This architecture is ideally suited to parsing inputs that are not fully understood or do not conform to a fixed grammar: the architecture warns about unrecognised inputs and other errors, but continues parsing as best it can, allowing the developer of a new parser to decide which deficiencies are most important and require immediate attention, rather than being forced to fix the first error that arises. The architecture is designed to enable the users of a parser to easily extend it to parse their particular inputs, without requiring much work or a high level of understanding of the parsing process and the parser's internal workings.

This architecture is the basis of Postfix Log Parser (PLP), a program that parses Postfix log files and places the resulting data into a Structured Query Language (SQL) database for later analysis. The gathered data can be used

to optimise current anti-spam defences, to produce statistics showing how effective each technique in use is, or to provide a baseline to test new anti-spam measures against. Numerous other uses are possible for such data: improving server performance by identifying troublesome destinations and reconfiguring appropriately; identifying regular high volume uses (e.g. customer newsletters) and restricting those uses to off-peak times or providing a dedicated service for them; detecting virus outbreaks that propagate via mail; billing customers on a shared server. Preserving the raw data enables a multitude of uses far beyond those conceived of by the author.

1.1 Thesis Layout

Chapter 2 provides background information useful in understanding SMTP, Postfix, and the motivation behind the project.

Chapter 3 reviews the previously published research in this area and other Postfix log file parsers, discussing why they were deemed unsuitable for the task, including why they could not be improved or expanded upon.

Chapter 4 describes the parser architecture developed for this project, beginning with an overview, then describing each of the components of the architecture in detail. This chapter concentrates on the abstract, theoretical, implementation-independent aspects of the architecture; discussion of the practical aspects is deferred until chapter 5.

Chapter 5 documents PLP, the parser based on the architecture described in chapter 4. The practical difficulties of implementing each of the components of the architecture are described, accompanied by the many complications encountered when parsing Postfix log files, and other details of the implementation.

Chapter 6 evaluates PLP's efficiency, exploring the various optimisations implemented in the parser and the effect they have. It also discusses the coverage achieved by PLP over 93 log files, with separate sections for the number of log lines correctly recognised, and the number of connections and mails whose journey through Postfix was correctly reconstructed.

Chapter 7 contains the conclusion of the thesis.

The bibliography (appendix A) contains references to the resources used in designing the architecture and writing PLP.

Appendix B provides a glossary of terms used in the thesis.

Appendix C contains a list of acronyms used in the thesis; uncommon acronyms will have an entry in the glossary too.

Appendix D provides a brief description of each Postfix daemon.

1.2 Previously Published Work

Portions of chapters 1 and 4–6 have previously been published at an international conference [3], and later reprinted in a journal [4]. Publication of the conference paper was supported by Science Foundation Ireland RFP 05/RF/CMS002.

Chapter 2

Background

This chapter provides background information helpful in understanding the remainder of this thesis. It begins with a discussion of the motivation underlying the project, followed by a brief introduction to Simple Mail Transfer Protocol (SMTP), and finishes with a longer introduction to Postfix, concentrating on anti-spam restrictions and policy servers.

2.1 Motivation

This work is part of a larger project to optimise a mail server's Postfix-based anti-spam restrictions, generate statistics and graphs, and provide a platform on which new restrictions can be developed and evaluated to determine if they are beneficial in the fight against spam. The program written for this project, Postfix Log Parser (PLP), parses Postfix log files and populates an Structured Query Language (SQL) database with the data gleaned from those log files, providing a consistent and simple view of the log files that future tools can utilise. The gathered data can be used to optimise current anti-spam measures, to provide a baseline to test new anti-spam measures against, or to produce statistics showing how effective those measures are.

Determining the ten Postfix restrictions that reject the highest number of delivery attempts is a short example of the analysis possible using data from the database:

```
SELECT name, description, restriction_name, hits_total
    FROM rules
    WHERE action = 'DELIVERY_REJECTED'
    ORDER BY hits_total DESC
    LIMIT 10;
```

If the database supports sub-selects (where the results of one query are used as a parameter in another), percentages can be obtained for the top ten restrictions using the query in figure 2.1, producing output similar to table 2.1 on the next page.

Figure 2.1: Sample SQL query showing the percentage of delivery attempts rejected by the top ten restrictions

```
SELECT restriction_name, hits_total,
        (hits_total * 100.0 /
            (SELECT SUM(hits_total)
                FROM rules
                WHERE action = 'DELIVERY_REJECTED'
            )
        ) || '%' AS percentage
    FROM rules
    WHERE action = 'DELIVERY_REJECTED'
    ORDER BY hits_total DESC
    LIMIT 10;
```

Another example is determining which restrictions are not effective: the query in figure 2.2 on the next page shows which restrictions rejected fewer than 100 delivery attempts in the last log file parsed, and the percentage of total rejections in that log file that each of those restrictions represents.

The sample database queries yield summary statistics about the efficiency of anti-spam techniques. Analysis of this kind is much easier to perform when the data source is a database than when trying to directly analyse log files.

Table 2.1: Sample output from an SQL query showing the percentage of delivery attempts rejected by the top ten restrictions

Restriction	Number of rejections	Percentage of total rejections
reject_unlisted_recipient	2338352	44.255%
reject_unlisted_sender	864180	16.355%
reject_rbl_client SpamHaus SBL-XBL	620039	11.734%
check_policy_service greylisting	497247	9.410%
reject_non_fqdn_hostname	374657	7.090%
reject_unknown_client	187674	3.551%
reject_rbl_client SpamHaus Zen	152409	2.884%
reject_unknown_sender_domain	62958	1.191%
check_recipient_access	61359	1.161%
reject_rbl_client DSBL	30443	0.576%

All percentages in table 2.1 are exclusive, i.e. the first restriction's number of rejections does not include the second restriction's number of rejections.

Figure 2.2: Sample SQL query showing ineffective restrictions

```
SELECT name, description, restriction_name, hits,
        (hits * 100.0 /
            (SELECT SUM(hits)
                FROM rules
                WHERE action = 'DELIVERY_REJECTED'
            )
        ) || '%' AS percentage
    FROM rules
    WHERE action = 'DELIVERY_REJECTED'
        AND hits < 100
    ORDER BY hits ASC;
```

2.2 Simple Mail Transfer Protocol (SMTP)

Simple Mail Transfer Protocol (SMTP) is the protocol used for transferring mail between the sending and receiving Mail Transfer Agent (MTA). It is a simple, human readable, plain text protocol, making it quite simple to test and debug problems with it. A detailed description of SMTP is beyond the scope of this thesis: the original protocol definition is in Request For Comments (RFC) 821 [28], later superceded by RFC 2821 [25]. Despite the simplicity of the protocol, many viruses and spam sending programs fail to implement it properly, so requiring strict adherence to the protocol specification is beneficial in protecting against spam and viruses.[1] A typical SMTP conversation resembles the following (the lines starting with a three digit number are sent by the server, all other lines are sent by the client):

```
220 smtp.example.com ESMTP
HELO client.example.com
250 smtp.example.com
MAIL FROM: <alice@example.com>
250 2.1.0 Ok
RCPT TO: <bob@example.com>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
Message headers and body sent here.
.
250 2.0.0 Ok: queued as D7AFA38BA
QUIT
221 2.0.0 Bye
```

[1] Originally all mail servers adhered to the principle of *Be liberal in what you accept, and conservative in what you send* from RFC 760 [27], but unfortunately that principle was written in a friendlier time. Given the deluge of spam that mail servers are subjected to daily, a more appropriate maxim could be: *Require strict adherence to relevant RFCs; implement the strongest restrictions you can; relax the restrictions and adherence only when legitimate mail is impeded.* It is neither as friendly nor as catchy, but it more accurately reflects the current circumstances.

An example deviation from the protocol:

```
220 smtp.example.com ESMTP
HELO client.example.com
250 smtp.example.com
MAIL FROM: Alice in Distribution alice@example.com
501 5.1.7 Bad sender address syntax
RCPT TO: Bob in Sales/Marketing bob@example.com
503 5.5.1 Error: need MAIL command
DATA
503 5.5.1 Error: need RCPT command
Message headers and body sent here.
502 5.5.2 Error: command not recognized
```

This example client is so poorly written that not only does it present the sender and recipient addresses improperly, it ignores the error messages returned by the server and carries on regardless. Many spam and virus sending programs have serious deficiencies; unfortunately, newer programs tend to be written by competent programmers or send mail using well written programs (e.g. Postfix or Sendmail on Unix hosts, Microsoft Outlook on Windows hosts). Traditionally a mail server would have done its best to deal with deficient clients, with the intention of accepting as much mail sent to its users as possible, e.g. by accepting sender or recipient addresses that were not enclosed in <>, or by ignoring the absence of a HELO command.[2] Given the volume of spam sent daily, this liberal approach is no longer viable.

2.3 The Postfix Mail Transfer Agent

Postfix is a MTA with the following design aims (in order of importance): security, flexibility of configuration, scalability, and high performance. It features extensive optional anti-spam restrictions, allowing an administrator

[2]The HELO command is the first command sent by the client in the SMTP conversation; it is required by the protocol, but its original purpose was to prevent mail loops by detecting a server trying to deliver mail to itself, so some implementations omitted it.

to employ those restrictions which they judge suitable for their server's needs, rather than a fixed set chosen by Postfix's author. These restrictions can be selectively applied, combined, and bypassed on a per-client, per-recipient, or per-sender basis, allowing different levels of stricture and permissiveness. Administrators can supply their own rejection messages to make it clear to senders why their mail was rejected. Policy servers (soon to be described in §2.3.2 [p. 26]) provide a simple way to write new restrictions without having to modify Postfix's source code. Unfortunately this flexibility has a cost: the complexity of Postfix's log files. Although it is usually relatively simple to use standard Unix text processing utilities to determine the fate of an individual mail, sometimes it can be quite difficult. For most mails the journey through Postfix is simple and brief, but the remainder can be quite complex (see §5.7 [p. 107] for details).

Postfix's design follows the Unix philosophy of *"Write programs that do one thing and do it well"* [30], and it is separated into multiple components that each perform one of the tasks required of an MTA, e.g. receive mail, send mail, deliver mail to a user's mailbox; full details can be found in [16]. Postfix's design is strongly influenced by security concerns: those components that interact with other hosts are not privileged,[3] so bugs in those components will not give an attacker extra privileges; those components that are privileged do not interact with other hosts, making it much more difficult for an attacker to exploit any bugs that may exist in those components.

2.3.1 Mixing and Matching Postfix Restrictions

Postfix restrictions are documented fully in [19, 20, 21]; the following is only a brief introduction.

Postfix uses one restriction list (containing zero or more restrictions) for each stage of the SMTP conversation: client connection, HELO command, MAIL FROM command, RCPT TO commands, DATA command, and end of data. The appropriate restriction list is evaluated for each stage, though by

[3]Privilege means the power to perform actions that are limited to the administrator, and are not available to ordinary users.

default the restriction lists for client connection, HELO, and MAIL FROM commands will not be evaluated until the first RCPT TO command is received, because some clients do not deal properly with rejections before this stage; a benefit of this delay is that Postfix has more information available when logging rejections. Each restriction is evaluated to produce a result of *reject*, *permit*, *dunno*, or the name of another restriction to be evaluated; other less commonly used results are possible as described in [5]. The meaning of *permit* and *reject* is obvious; *dunno* means to stop evaluating the current restriction and continue processing the remainder of the restriction list, allowing exceptions to more general rules. The administrator can define new restrictions as a list of existing restrictions, allowing arbitrarily long and complex user-defined sequences of lookups, restrictions, and exceptions.

Postfix uses simple lookup tables to make decisions when evaluating some restrictions, e.g.

```
check_client_access cidr:/etc/postfix/client_access
```

check_client_access The name of the restriction to evaluate.

cidr The type of the lookup table.

/etc/postfix/client_access The file containing the lookup table.

The restriction `check_client_access` checks if the IP address of the connected client is found in the specified table and returns the associated result if found; the method of searching the file is dependant on the type of the lookup table [18]. Other restrictions determine their result by consulting external sources, e.g.

```
reject_rbl_client dnsbl.example.com
```

checks the DNS Blacklist (DNSBL) `dnsbl.example.com` and rejects the command if the client's IP address is listed.

The configuration example below shows how to require that all machines on the local network except for the web server authenticate before sending mail; the web server is exempt because the legacy applications running on it lack authentication support. The restriction list is evaluated from top to bottom: `permit_sasl_authenticated` permits authenticated

clients; an exception is made for the web server in the `check_client_access` check, and then other local machines are rejected, because to continue past `permit_sasl_authenticated` the client must not have authenticated.

```
main.cf:
smtpd_client_restrictions =
  <other restrictions>
  permit_sasl_authenticated,
  check_client_access /etc/postfix/allow_webserver.cidr,
  <other restrictions>

/etc/postfix/allow_webserver.cidr:
  192.0.2.80/32   dunno
  192.0.2.0/24    reject "Please authenticate to send mail"
```

That example also shows how to supply a custom rejection message. When the result of a lookup is the name of another restriction Postfix will evaluate that restriction; this allows restrictions to be chosen based on the client IP address, client hostname, HELO hostname, sender address, or recipient address. For example, the administrator may require that all clients on the local network have valid DNS entries, to prevent people sending mail from unknown machines; one example of how to achieve this is:

```
main.cf:
smtpd_client_restrictions =
  <other restrictions>
  check_client_access /etc/postfix/require_dns_entries.cidr,
  <other restrictions>

/etc/postfix/require_dns_entries.cidr:
  192.0.2.0/24    reject_unknown_client_hostname
```

Postfix tries to protect the administrator from misconfiguration in as far as it reasonably can, e.g. the restriction `check_helo_mx_access` cannot cause

a mail to be accepted, because the parameter it checks (the hostname given in the HELO command) is under the control of the remote client. Despite this, it is possible for the administrator to make catastrophic mistakes, e.g. rejecting all mail — the administrator must be cognisant of the effects their configuration changes will have. This is similar to one of UNIX's design philosophies: *"UNIX was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things"* [30].

2.3.2 Policy Servers

A policy server [20] is an external program consulted by Postfix to determine the fate of an SMTP command. The policy server is given state information by Postfix (sample state information is shown in table 2.2 on the following page) and returns a result (reject, permit, dunno, or a restriction name) as described in §2.3.1 [p. 23]. A policy server can perform more complex checks than those provided by Postfix, such as allowing addresses associated with the payroll system to send mail on the third Tuesday after pay day only, to help prevent problems from phishing mails using faked sender addresses. For example, a phishing mail might pretend that the payroll system had a disastrous disk failure, and until the server is replaced all salary payments will have to be processed manually, so please reply to this mail with your name, address, and bank account details; the criminal can then use any details sent to him to help with identity theft.

Some widely deployed policy servers:

- Checking if the client satisfies a domain's Sender Policy Framework (SPF) records, `http://www.openspf.org/` (last checked 2009/04/21). SPF records specify which mail servers are allowed to send mail using sender addresses from a particular domain. The intention is to reduce spam from faked sender addresses, backscatter, and joe jobs. There has been considerable resistance to SPF because it breaks or vastly complicates some commonly-used features of SMTP, e.g. forwarding mail from one domain to another when a user moves.

Table 2.2: Example attributes sent to policy servers, taken from [20]

Attribute name	Attribute value
request	smtpd_access_policy
protocol_state	RCPT
protocol_name	SMTP
helo_name	some.domain.tld
queue_id	8045F2AB23
sender	foo@bar.tld
recipient	bar@foo.tld
recipient_count	0
client_address	1.2.3.4
client_name	another.domain.tld
reverse_client_name	another.domain.tld
instance	123.456.7

- Greylisting, `http://www.greylisting.org/` (last checked 2009/04/21), is a technique that temporarily rejects a delivery attempt when the tuple of

  `(sender address, recipient address, remote IP address)`

  has not been seen before; on second and subsequent delivery attempts from that tuple the mail will be accepted. This blocks spam from some senders because maintaining a list of failed addresses and retrying after a temporary failure is uneconomical for a spam sender, but a legitimate mail server must retry deliveries that fail temporarily. Sadly spam senders are using increasingly complex and well written programs to distribute spam, frequently using an ISP provided SMTP server from a compromised machine on the ISP's network. Greylisting will slowly become less effective as spam senders adapt, but it does block a large percentage of spam mail at the moment; the most effective restrictions from the 93 log files used when generating the results in §6 [p. 129] are shown in table 2.1 [p. 20]. That table shows that greylisting is worth using at the moment, particularly when you take into account its position as the final restriction that a mail must overcome in the

configuration used on the mail server that generated the log files: on that server greylisting only takes effect for mails that have passed all other restrictions. Some problems may be encountered when using greylisting: some servers fail to retry after a temporary failure, or legitimate mail may be delayed, particularly when coming from a pool of servers.

- Scoring systems such as postfwd, `http://postfwd.org/` (last checked 2009/04/21), perform tests on features of the delivery attempt (e.g. IP address, sender address), incrementing or decrementing a score based on the results; if the final score is higher than a threshold the delivery attempt is rejected. The administrator must manually whitelist clients if they are to bypass a Postfix restriction, whereas using a threshold that requires a delivery attempt to hit several scored restrictions will allow delivery attempt that would be rejected by a boolean restriction.

2.4 Summary

This chapter has provided background information useful in understanding this thesis, starting with the motivation behind the project, continuing with an introduction to SMTP, and finishing with Postfix, its anti-spam restrictions, and its support for policy servers.

Chapter 3

State of the Art Review

Ten Postfix log file parsers were tested at the start of this project, with the hope of finding a suitable parser to build upon, rather than writing one from scratch. It was quite difficult to find ten parsers to review, and the functionality offered by those parsers ranges from quite basic to more developed, depending on the needs of the parser's author.

It was hoped to reuse an existing parser rather than writing one from scratch, but the existing parsers considered were rejected for one or more reasons. The effort required to adapt and improve an existing parser was judged to be greater than the effort to write a new one, because the techniques used by the existing parsers severely limited their potential: some ignored the majority of log lines, parsing specific log lines accurately, but without any provision for parsing new or similar log lines; others sloppily parsed the majority of log lines, but were incapable of distinguishing between log lines within one category, e.g. not distinguishing between different anti-spam techniques causing delivery attempts to be rejected. The first parser reviewed [11] is the only previously published research in this area that the author could find; that research aims to show that providing the data from log files in a more accessible form is helpful to systems administrators.

The ten parsers have been compared and contrasted with Postfix Log Parser (PLP), this project's parser, to show how much effort would have been required to use one of those parsers to fulfil the aims and requirements

of this project. It is important to compare and contrast newly developed algorithms and parsers against those already available, to accurately judge what improvements, if any, are delivered by the newcomers. Table 3.1 [p. 52] summarises the results of this review.

Some important differences exist between PLP and most or all of the parsers reviewed in this chapter:

1. None of the reviewed parsers perform the kind of advanced parsing required for this project, or deal with the complications described in §5.7 [p. 107]. Some correlate log lines by queueid, but none correlate log lines by Process Identifier (pid) (§5.7.1 [p. 109]).

2. Only PLP enables parsing of new log lines without extensive and intrusive modifications to the parser; the architecture enabling this is documented in §4 [p. 53].

3. The reviewed parsers all produce a report of varying complexity and detail, whereas PLP does not: it extracts data and leaves generation of reports to other programs. Using a Structured Query Language (SQL) database simplifies the process of generating such reports (discussed in §5.3.1 [p. 77]); some sample queries are given in §2.1 [p. 18]. The parser developed for this project is designed to enable much more detailed log file analysis than other parsers by providing a stable platform for subsequent programs to build upon.

4. Most of the reviewed parsers silently ignore log lines they cannot handle, whereas PLP warns about every single log line it fails to recognise. The exception is AWStats, which outputs the percentage of log lines it was unable to parse, but does not output the log lines themselves.

5. A minor difference is that most parsers do not handle compressed log files; both PLP and Splunk handle them transparently, without user intervention; Sawmill and Lire can be configured to support compressed log files, but Sawmill exhibits a dramatic increase in parsing time when doing so. Support for reading compressed log files is quite helpful,

because it dramatically reduces the disk space required to store historical log files.

6. Most of the reviewed parsers do not distinguish between different delivery attempt rejections, so they cannot be used to determine the success rate of different anti-spam techniques. The exception is Pflogsumm, which provides a summary of why delivery attempts were rejected.

Each of the reviewed parsers was tested with the three months (93 days) of contiguous log files described in §6.1 [p. 130].

The data extracted by PLP is documented in §5.3.3 [p. 82] and §5.3.4 [p. 83], but for convenience the list is repeated here: client and server IP address and hostname, HELO hostname, queueid, start time, end time, Simple Mail Transfer Protocol (SMTP) code, enhanced status code, sender, recipient, size, message-id, delay, and delays. Note: PLP correlates log lines by both queueid and pid (§5.7.1 [p. 109]), and it stores each mail's queueid, but it does not store each connection's pid, because a use has not been identified for that data.

3.1 Log Mail Analyser

There appears to be only one prior published paper about parsing Postfix log files: *Log Mail Analyzer: Architecture and Practical Utilizations* [11]. The aim of Log Mail Analyzer (LMA) is quite different from PLP: it attempts to present correlated data from log files in a form suitable for a systems administrator to search using the myriad of standard Unix text processing utilities already available. It produces a Comma-Separated Value (CSV) file and either a MySQL or Berkeley DB database. Hardly any documentation is provided with LMA, but some documentation is available in [11]. Studying the source code is informative, though this author had some difficulty because the authors of LMA wrote in Italian.

CSV CSV is a very simple format where each record is stored in a single line, with fields separated by a comma or other punctuation symbol.

Problems with CSV include the need to escape separators in the data stored, providing multiple values for a field (e.g. multiple recipients), and adding new fields. CSV does not have a standard mechanism to document the fields or the separator, unlike SQL where every database includes a schema naming the fields and the type of data they store (e.g. integer, text, timestamp). The CSV record format used by LMA is not documented in [11], but the output file contains a comment in Italian giving the format:

```
# Timestamp|Nome Client|IP Client|IP Server|From|To
  |Status|Size
```

LMA treats CSV lines starting with `#` as comments, but not all CSV parsers will.

Berkeley DB Berkeley DB only supports storing simple **(key, value)** pairs, unlike SQL databases that store arbitrary tuples. In LMA's main table the key is an integer referred to by secondary tables, and the value is a CSV line containing all of the data for that row. The secondary by-sender, by-recipient, by-date, and by-IP tables use the sender/recipient/date/IP address as the key, and the value is a CSV list of integers referring to the keys in the main table. This effectively re-implements SQL foreign keys, but without the functionality offered by even the most basic of SQL databases, e.g. joins, ordering, searches. It also requires custom code to search on a combination of attributes, and the authors of LMA did provide some simple reports: IP-STORY, FROM-STORY, DAILY-EMAIL, and DAILY-REJECT (all described later). Berkeley DB appears to be the least useful of the three output formats: it does not provide the functionality of a basic SQL database, and unlike CSV files it cannot be used with standard Unix text processing tools.

MySQL MySQL is a widely used, open source, relational database. LMA's MySQL support was not tested because the database schema used by LMA is not documented, so the required database could not be created.

Whether a MySQL or Berkeley DB database is chosen in addition to the CSV output, LMA stores the following data: time and date of the log line, client hostname and IP address, server IP address, sender and recipient addresses, SMTP code, and size (for accepted mails only). Unlike PLP it does not store the server hostname, HELO hostname, queueid, start and end times, timestamps for each log line, enhanced status code, delivery delays, or message-id (for accepted mails only). Handling of multiple recipients, SMTP codes, or remote servers[1] is not explained; experimental observation shows that multiple records were added when a mail had multiple recipients, but the records are not associated or linked in any way, and presumably the same approach was taken when there were multiple destination servers.

LMA requires major changes to the parser code to parse new log lines or to extract additional data. The code is structured as a long series of blocks that each handle all log lines matching a single Regular Expression (regex), so parsing new log lines requires modifying an existing regex or carefully inserting a new block in the correct place; extracting extra data would require modifying multiple blocks, regexes, or both. Regular Expressions are a compact, powerful method of specifying patterns that describe a set of strings. The regex `aa*b*b` describes a set of strings, all of which start with `a`, followed by any number of `a`, then any number of `b`, and finish with `b`; the strings `ab`, `aaaaaaaab`, and `aaabbbbb` are members of that set, whereas the strings `abba`, `abcd`, and `qwerty` are not. A string can be checked against a regex to determine if the string is a member of the set of strings described by that regex.

LMA does not deal with any of the complications discussed in §5.7 [p. 107], except for correlating log lines by queueid; it cannot correlate most rejected delivery attempts because it does not correlate log lines by pid. It does not differentiate between different types of rejections, so it is not suitable for the purposes of this project; the data about which restriction caused the rejection is discarded, whereas the main goal of this project is to retain that information to aid optimisation and evaluation of anti-spam techniques. LMA

[1] A single mail may be sent to multiple remote servers if it was addressed to recipients in different domains, or Postfix needs to try multiple servers for one or more recipients.

fails to parse Postfix log files generated on Solaris hosts because the fields automatically prepended to each log line differ from those prepended on Linux hosts; log files from Solaris hosts (and possibly other operating systems) thus require pre-processing before parsing by LMA. The 93 pre-processed test log files were parsed without complaint by LMA, although it produced 32 entries in the output CSV file for every rejection in the input log files; it also missed some 40% of delivered mail. Once these deficiencies were discovered the author did not spend any more time checking the results.

LMA does provide some simple reports: IP-STORY, FROM-STORY, DAILY-EMAIL and DAILY-REJECT. These reports search the Berkeley DB files for matching records: the first three produce CSV lines for the specified client IP address, sender address, or date respectively. DAILY-REJECT initially failed with an error message from the Perl interpreter;[2] after correcting the errors in the code it worked, producing CSV lines for the specified day where the SMTP code signifies a rejection. All of these reports are extremely simple to produce from the CSV file using the standard Unix tool `awk`; the most complicated, DAILY-REJECT, is merely:

```
awk -F\| 'BEGIN { previous = "" };
    $1 ~ /2007-01-26/ && $7 != "250" && $0 != previous
    { print $0; print " "; previous = $0; }' lma_output.txt
```

Notes about the command above:

- It outputs a line containing only a single space after each matching record, to accurately replicate the output of DAILY-REJECT.
- DAILY-REJECT considers all SMTP codes except "250" to be rejections; this includes invalid SMTP codes such as `0` and `deferred`, so the awk command does too. These invalid SMTP codes are most likely present because of incorrect parsing by LMA.

[2]The error messages were:

```
Undefined subroutine &main::LIST called at queryDB.pl line 372.
Undefined subroutine &main::EXTRACT_FROM_DB called at queryDB.pl line 379.
```

- LMA produces 32 lines in its CSV file for every single line it should have produced; the command above suppresses duplicate sequential lines. DAILY-REJECT produces the correct number of output lines, probably because it searches the Berkeley DB database rather than the CSV file.

The output from DAILY-REJECT and the `awk` command is not exactly the same; the author did not spend substantial time attempting to rectify these differences.

1. The output from DAILY-REJECT is missing some records that are present in the CSV file; this may indicate differences between the data stored in the CSV and Berkeley DB files.

2. Some records output by DAILY-REJECT are truncated: they are missing the last "|" that separates fields and also the newline following it, so the line containing only a single space is concatenated with the record.

In summary, LMA appears to be a proof of concept, written to demonstrate the point of their paper, that making the information contained in log files available in an accessible fashion is useful to systems administrators, rather than a program intended to be useful in a production environment.

3.2 Pflogsumm

> *pflogsumm is designed to provide an over-view of Postfix activity, with just enough detail to give the administrator a "heads up" for potential trouble spots.*
> `http://jimsun.linxnet.com/postfix_contrib.html`
> Last checked 2008/11/23.

Pflogsumm produces a report designed for troubleshooting rather than in-depth analysis. It does not support saving any of the data it extracts from log files, and it does not extract any data that it does not require to produce

its report: the HELO hostname, queueid, start and end times, timestamps for each log line, or message-id. Both the parsing and reporting are difficult to extend because it is a specialised tool, unlike the easily extensible design of PLP. It does not correlate log lines by queueid or pid, and does not need to deal with the complications encountered during this project (§5.7 [p. 107]). Pflogsumm produces a useful report, and successfully parsed the 93 log files it was tested with. The results it reported were not verified in detail, but it did not report any errors, and has an excellent reputation amongst Postfix users. Pflogsumm has many options to include or exclude certain sections of the report, all clearly documented; by default its report includes the following:

- Total number of mails accepted, delivered, and rejected. Total size of mails accepted and delivered. Total number of sender and recipient addresses and domains.
- Per-hour averages and per-day summaries of the number of mails received, delivered, deferred, bounced, and rejected.
- For received mail: per-domain totals for the number of mails sent, number of mails deferred, average delay, maximum delay, and bytes delivered. For received mail: per-domain totals for size and number of mails received.
- Number and size of mails sent and received for each address.
- Summary of why mail delivery was deferred or failed, why mails were bounced, why mails were rejected, and warning messages from Postfix.

Pflogsumm is the only reviewed parser that distinguishes between different rejected delivery attempts, an important requirement for this project.

3.3 Sawmill Universal Log File Analysis and Reporting

Sawmill is a Postfix log analyzer (it also support 818 other log formats). It can process log files in Postfix format, and gener-

> *ate dynamic statistics from them, analyzing and reporting events. Sawmill can parse Postfix logs, import them into a SQL database (or its own built-in database), aggregate them, and generate dynamically filtered reports, all through a web interface. Sawmill can perform Postfix analysis on any platform, including Window, Linux, FreeBSD, OpenBSD, Mac OS, Solaris, other UNIX, and more.*
> `http://www.thesawmill.co.uk/formats/postfix.html`
> Last checked 2008/11/23.

Sawmill is a general purpose commercial product that parses 818 log file formats (as of 2008/11/23) and produces reports from the extracted data. Its data extraction facilities (described later) are too limited to save enough data for the purposes of this project: although it can extract three different sets of data from Postfix log files, they are not linked in any way. The documentation does not suggest that Sawmill correlates log lines by either queueid or pid, or deals with the other difficulties documented in §5.7 [p. 107].

Sawmill has three different Postfix log file parsers, extracting three different sets of data:

1. `http://www.thesawmill.co.uk/formats/postfix.html`
   (last checked 2008/11/23). Fields extracted: from, to, server, uid, relay, status, number of recipients, origin hostname, origin IP address, and virus. It also counts the number of and total size of all mails delivered. The fields `server`, `uid`, `relay`, and `virus` are not explained in the documentation: `server` is probably the hostname or IP address of the server the mail is delivered to; `relay` might be the delivery method: SMTP, local delivery, or Local Mail Transfer Protocol (LMTP); `uid` might be the uid of the user submitting mail locally. Postfix does not do any form of virus checking itself, so the `virus` field is a mystery.

2. `http://www.thesawmill.co.uk/formats/postfix_ii.html`
   (last checked 2008/11/23). Fields extracted: from, to, RBL list, client hostname, and client IP address. It also counts the number and total size of all mails delivered.

3. `http://www.thesawmill.co.uk/formats/beta_postfix.html` (last checked 2008/11/23). Fields extracted: from, to, client hostname, client IP address, relay hostname, relay IP address, status, response code, RBL list, and message-id. It also counts the number and size of all mails delivered, processed, blocked, expired, and bounced.

Even if the three data sets were combined Sawmill would extract less data than PLP: it omits the HELO hostname, queueid, enhanced status code, delivery delays, and start and end times. Sawmill does not extract any data about rejections except when the rejection is caused by a DNS Blacklist (DNSBL) check (`RBL list` in the list of fields).

The source code is only available in an encrypted form, to support people who wish to use Sawmill on operating systems or machine architectures the company do not provide executables for. Sawmill is quite expensive, requiring a €100 + VAT licence per report, with discounts available when buying multiple licences (correct as of 2008/11/23); in contrast, PLP is free to use and the code is freely available. Sawmill is supplied with thorough and well written documentation; everything the author looked for was documented, except for the MySQL database schema and some details of the data extracted by the parser, e.g. what the virus field stores. A commercial version of MySQL is required because of MySQL licensing restrictions, but Sawmill's documentation explains why and includes instructions on how to compile Sawmill so that it can use a non-commercial version of MySQL (this was not attempted during the review process).

Sawmill's web interface supports searching on any combination of the fields it extracts, and all searches produced accurate results. The interface for searching is neither as simple to use nor as informative as the interface provided by Splunk (see §3.4 on the next page). However, the administrative interface is much easier to use than Splunk's: it took only five minutes to start parsing all of the log files in a directory.

When tested with the 93 test log files it performed adequately, though the rate it processed log files at did slowdown noticeably as it progressed. Sawmill supports reading compressed log files but it exhibits a dramatic slow down

when doing so: it took six hours to parse the first half of the log files, and twelve hours to parse the next third; after twenty four hours spent parsing the remaining sixth it crashed due to lack of disk space. On the second parsing attempt the log files were uncompressed beforehand and parsing took eight hours.

In summary, Sawmill suffers from supporting so many types of log files: it is probably much more useful when parsing log files where each log line is self-contained (e.g. web server log files), rather than log files where data is spread across multiple log lines. It is not suitable as a base for this parser, because the source code made available is encrypted and not intended for modification; in addition the architecture would probably need to be overhauled or replaced to deal with correlating log lines.

3.4 Splunk

> *Splunk is IT Search. Search and navigate IT data from applications, servers and network devices in real-time. Logs, configurations, messages, traps and alerts, scripts, code, metrics and more. If a machine can generate it — Splunk can eat it. It's easy to download and use and it's very powerful.*
> `http://www.splunk.com/`
> Last checked 2008/11/23.

Splunk aims to index all of an organisation's log files, providing a centralised view capable of searching and correlating diverse log sources. The web interface provides search functionality, generating statistics and graphs in real time, a facility not provided by PLP. Splunk allows quite complicated searches, based on the fields it extracts (described later) or the full text of the log line, though it is not possible to search on partial words. Searches can be saved for reuse; saved searches can be run periodically and the results mailed to a recipient or sent to an external program for further processing. The author was unable to save searches, possibly because of limitations in the free version, and so was unable to examine the format of the data. The web interface is

optimised for interactive use rather than automated queries, and it does not appear to be possible to write independent tools to utilise the Splunk database, whereas PLP provides the database and leaves it to the user to utilise it without limit or restriction. Some additional Postfix reports are supposedly available on `http://www.splunkbase.com/` (last checked 2009/04/21), but the author was unable to find any Postfix reports, or indeed reports for any other log file types: every category was empty, even those that the web site claimed had numerous reports available. Many types of graphs can be generated, though all except the bubble and heatmap graphs are variations of a bar or pie chart. Drilling down through the graphs to select a portion of the data is simple and intuitive, e.g. select the hour with the largest number of events, then select a particular host, and finally a specific sender address. All searches performed using the indexed data returned reasonable results. The full power of SQL is available when searching the data extracted by PLP, allowing the user to search on arbitrarily complicated conditions.

The web interface is quite attractive and simple to use when searching, but as an administrator it seems unnecessarily difficult to perform simple tasks. When testing Splunk it took roughly 30 minutes to figure out how to add a single log file to be indexed for later searching, with the added downside that the log file was copied into a spool directory before indexing, doubling the disk space usage. The next test was to index all the log files in a particular directory, but after three hours, reading all the available documentation, and numerous futile attempts, the author was still unable to index all the log files in a directory using the web interface. Using Splunk's command line interface rather than the web interface was more successful: the command "`splunk find logs` *log-directory*" added 40 of the 93 log files to the queue for indexing. Further attempts enqueued the same 40 log files, without explaining why the others were excluded.[3] The command did not have an option to ensure the log files would be processed in the order they were created, though such an option may be neither necessary nor beneficial

[3]The log files that were added multiple times appear to have been indexed once only; presumably Splunk keeps track of the log files it has indexed and discards requests to index log files for a second time. This may or may not be a useful feature for PLP.

with Splunk. Subsequently the author was successful in adding a single log file at a time using the command "`splunk add tail` *filename*", and then a simple loop using that command was enough to add all the desired log files. Splunk will periodically check all indexed log files for updates unless they are manually removed from its list; this may or may not be useful behaviour. Splunk did not appear to have any difficulty in indexing the log files once they had been successfully added to its queue. PLP parses the log files it is instructed to parse, in the order they are given; periodic parsing of log files is a task an administrator can easily achieve with `cron(8)` and `logrotate(8)`.

Copious documentation is made available on `http://www.splunk.com/` (last checked 2009/04/21), but the abundance of material and lack of organisation makes it hard to find the topic being sought, and searches confusingly tended to return results from old documentation rather than new. In general the documentation appears to have been written by someone intimately acquainted with the software, who has difficulty understanding how a newcomer would approach tasks or the questions they would ask.

Splunk supports reading compressed log files without any extra configuration by the user, like PLP. The free version of Splunk limits the volume of data indexed per day to 500MB, though a trial Enterprise licence is available that allows indexing of up to 5GB of data per day. In 2007, the cheapest licenced version cost $5000 plus $1000 support per annum, and limited the volume of data indexed per day to 500MB. Prices were removed from the Splunk website during 2008; now Splunk's sales team must be asked for a quote. Median log file size for the 93 log files used when evaluating PLP is 53.297 MB.

When parsing Postfix log files Splunk parses the standard syslog fields at the beginning of the log line, and extracts any `key=value` pairs occurring after the syslog prologue: to and from addresses, HELO hostname, and protocol (SMTP, LMTP, or Extended SMTP (ESMTP)). PLP extracts noticeably more data: client and server IP address and hostname, queueid, start and end times, timestamps for each log line, SMTP and enhanced status codes, delivery delays, and message-id. PLP does not make the full text of the log line available; a few minutes work could add this to PLP if desired, but it

would greatly increase the size of the database.

Splunk is a generic tool, so it lacks any Postfix-specific support over and above extracting the `key=value` fields from each log line; it makes no attempt to correlate log lines by queueid or pid, or to handle any of the other myriad complications discussed in §5.7 [p. 107]. Its source code is unavailable, so it could not be used as a base for this project, even if it fulfilled all the other requirements.

3.5 Isoqlog

> *Isoqlog is an MTA log analysis program written in C. It designed to scan qmail, postfix, sendmail and exim logfile and produce usage statistics in HTML format for viewing through a browser. It produces Top domains output according to Sender, Receiver, Total mails and bytes; it keeps your main domain mail statistics with regard to Days Top Domain, Top Users values for per day, per month and years.*
> `http://www.enderunix.org/isoqlog/`
> Last checked 2009/01/11.

Isoqlog's report lacks most of the information gathered by PLP: the data it extracts is limited to the number of mails sent by each sender, and it only reports on senders from the domains listed in its configuration file, making it impossible to produce complete reports. It ignores all log lines except those with today's date, so it is impossible to analyse historical log files, and testing with the 93 test log files was pointless. It does maintain a record of data previously extracted and newly extracted data is added to it; the format of the data store is undocumented. Almost no documentation is provided with Isoqlog, little more than installation instructions. It does not utilise rejection log lines in any way, so is unsuitable for the purposes of this project. Its parsing is completely inextensible, indeed is almost incomprehensible, relying on `scanf(3)`, unexplained fixed offsets, and low level string manipulation; it is the opposite end of the spectrum to PLP's parsing. It does not handle any

of the complications discussed in §5.7 [p. 107], does not gather the breadth of data required for this project, and ignores most of the log lines produced by Postfix.

3.6 AWStats

> *AWStats is a free powerful and featureful tool that generates advanced web, streaming, ftp or mail server statistics, graphically. This log analyzer works as a CGI or from command line and shows you all possible information your log contains, in few graphical web pages. It uses a partial information file to be able to process large log files, often and quickly. It can analyze log files from all major server tools like Apache log files (NCSA combined/XLF/ELF log format or common/CLF log format), WebStar, IIS (W3C log format) and a lot of other web, proxy, wap, streaming servers, mail servers and some ftp servers.*
> `http://awstats.sourceforge.net/awstats.mail.html`
> Last checked 2009/01/11.

AWStats can produce simple graphs from many different services' log files, but supporting numerous log file formats without special purpose code limits its functionality. The data it can extract from Postfix log files is limited in comparison to PLP: time2, email, email_r, host, host_r, method, url, code, and bytesd. No explanation for any of those fields is provided in the documentation at `http://awstats.sourceforge.net/docs/awstats_faq.html#MAIL` (last checked 2009/04/21), so the author could neither understand what data is extracted, nor determine what data is missing in comparison to PLP, which fully documents all the data it extracts. AWStats coerces Postfix log files into the log file format used by the Apache web server, for analysis by AWStats' HTTP log file parser. The converting parser only deals with a small portion of the log lines generated by Postfix, silently skipping those it cannot parse, and does not distinguish between different delivery attempt rejections; extending it to parse all log lines would be at least as much work as writing a new parser.

It does correlate log lines by queueid (not by pid), but it does not deal with any of the other complications described in §5.7 [p. 107]. AWStats supports saving data extracted from log files, but the format of the data store is not documented. It also supports reading compressed log files, but that was not tested.

When tested with the 93 test log files, AWStats reported that it parsed 9,240,075 (88.709%) of 10,416,129 log lines, skipping 1,176,050 (11.290%) corrupt log lines. However, the 93 test log files contain 60,721,709 log lines, so AWStats parsed only 15.217% of the log lines, declared that 1.936% were corrupt, and ignored the remaining 82.846%. The parsing results were not examined in detail or verified.

The graphs it produces give an overview of mails received for the last calendar month, showing:

- The number of mails accepted from each host.
- How many mails were received by each recipient.
- The average number of mails accepted by the server per-day and per-hour.
- A summary of the SMTP codes used when rejecting delivery attempts.

AWStats was not a suitable base for this project because it assumes that all log files can be rewritten to be compatible with web server log files, and will contain similar data; coercing Postfix log files into web server log files, without substantial data loss, would require fully parsing the Postfix log files without using AWStats, i.e. would require writing a separate parser anyway. It may be possible to use AWStats' graphing capabilities to generate reports, by generating input for AWStats from the database populated by PLP, but the author has not attempted that.

3.7 Anteater

> *The Anteater project is a Mail Traffic Analyser. Anteater supports currently the logformat produced by Sendmail and by Postfix. The*

> *tool is written in 100% C++ and is very easy to customize. Input, output, and the analysis are modular class objects with a clear interface. There are eight useful analyse modules, writing the result in plain ASCII or HTML, to stdout or to files.*
> `http://anteater.drzoom.ch/`
> Last checked 2009/01/11.

Anteater does not have any English documentation except for the quote above so it is impossible for this author to accurately comment on the analysis it performs. It did not run successfully when tested, and its parsing would certainly be out of date because Postfix has evolved considerably since this tool was last updated (2003/11/06). Because it neither ran successfully nor has documentation the author can read, a detailed review cannot be provided.

The Debian Linux distribution provides a translated manual page with the copy of anteater it distributes, so the author was at least able to run anteater with the correct arguments; sadly anteater produced zero for every statistic, presumably because it was unsuccessful in parsing the log lines.

3.8 Yet Another Advanced Logfile Analyser

> *yaala is a very flexible analyser for all kinds of logfiles. It uses parsers to extract information from a logfile, an SQL-like query language to relate the information to each other and an output-module to format the information appropriately.*
> `http://yaala.org/`
> Last checked 2009/01/11.

YAALA uses plugins to analyse log files and produce reports in HTML format. Using YAALA as a base for this project would have been as much work as starting from scratch, because both the input and output modules would need to be written specially; it might be more work to implement a parser within the constraints of YAALA rather than independently. YAALA supports storing previously gathered data using Perl's Storable module, so other Perl programs could use Storable to load, examine, and optionally

modify the data; PLP uses a well documented database that is accessible from most common programming languages. Information about how YAALA stores data was gleaned from the source code, because the format is undocumented and differs amongst plugins.

YAALA provides a Postfix parser that extracts the following fields from specific log lines:

Aggregations: count (not explained), bytes (sum of bytes transferred).

Keyfields: incoming_host, outgoing_host, date, hour, sender, recipient, defer_count, delay. Which date and hour are stored is not documented: start time, end time, delivery time, or another time?

YAALA's Postfix log file parser extracts some of the fields PLP does: for client and server it stores either the IP address or the hostname, not both; it omits the HELO hostname, queueid, SMTP and enhanced status codes, size of each accepted mail, start and end times, timestamps for each log line, and message-id. It extracts one piece of data that PLP does not: how many times delivery was deferred for each mail; this information can be calculated from the database populated by PLP if desired. Unlike PLP, YAALA does not maintain separate counters for different delivery attempt rejections, precluding the possibility of using the collected data for optimisation, testing, or understanding of restrictions. YAALA's Postfix log file parser does not deal with the complications explained in §5.7 [p. 107], except that it does correlate log lines by queueid (but not by pid).

YAALA provides a mini-language based on SQL that is used when generating reports; sample reports can be seen at `http://www.yaala.org/samples.html` (last checked 2009/04/21). Example query for HTTP proxy servers:

```
requests BY file WHERE host =~ Google
```

The mini-language is quite limited and cannot be used to extract data for external use, merely to create reports. Only data selected by the query will be saved in the data store; other data will be discarded, and removed from the data store if already present.

Testing YAALA was unsuccessful because all the queries produced a similar error message:

```
lib/Yaala/Data/Core.pm: Unavailable aggregation requested:
  ‘‘bytes’’. Returning 0.
```

The underlying reason for this is that YAALA only parsed 408 (0.11%) of 360,632 log lines in the first log file; it was not tested with the remainder of the 93 log files.

It might be possible to use PLP as a plugin with YAALA, perhaps with an intermediate plugin interfacing between the two, but YAALA's data store is insufficient for PLP's needs: PLP uses two separate tables, whereas YAALA assumes all data will reside in one structure; YAALA's querying mini-language might not deal successfully with data in separate structures. This approach has not been attempted by the author.

In summary, YAALA provides a Postfix log file parser that unsuccessfully attempts to parse only the most common Postfix log lines, provides reasonably flexible report generation from the limited data extracted, but has no facilities to extract data for use in other tools.

3.9 Lire

> *As any good system administrator knows, there's a lot more to keep track of in an active network than just webservers. Lire is hands down the most versatile log analysis software available today. Lire not only keeps you informed about your HTTP, FTP, and mail traffic, it also reports on your firewalls, your print servers, and your DNS activity. The ever growing list of Lire-supported services clearly outstrips any other software, in large part thanks to the numerous volunteers who have pioneered many new services and features. Lire is a total solution for your log analysis needs.*
>
> `http://logreport.org/lire.html`
>
> Last checked 2009/01/11.

Lire is a general purpose log file parser supporting many different types of log file. It takes a similar approach to YAALA, using plugins to parse different log file types. The data extracted by its Postfix log file parser is not clearly documented: *The email servers' reports will show you the number of deliveries and the volume of email delivered by day, the domains from which you receive or send the most emails, the relays most used, etc.*

Examining the source code reveals that Lire looks for `<key>=<value>` pairs in each log line, extracts them, and correlates the data by queueid (but not by pid). This approach will extract the following data: HELO hostname, queueid, SMTP code, sender and recipient addresses, and size of accepted mails. Lire misses the following fields extracted by PLP: client and server IP address and hostname, start and end times, enhanced status code, delivery delays, timestamps of each log line, and message-id.

Lire supports multiple output formats for the reports it generates (text, HTML, PDF, and Excel 95) but the reports do not appear to be customisable, and are not as detailed as Pflogsumm's; PLP does not produce any reports. Lire supports saving extracted data for later report generation, but the format of this data store is undocumented. PLP uses an SQL database to make accessing the extracted data as effortless as possible. In general, Lire has poor documentation.

Similar to AWStats and Logrep (§3.10 on the following page), Lire attempts to correlate log lines by queueid, but not by pid, so the complete list of recipients for each delivered mail should be available; it does not attempt to deal with the other complications described in §5.7 [p. 107]. When testing Lire on the 93 test log files it performed reasonably well: the numbers it reports appear accurate, and the subset verified by the author were correct. Its report provides summaries of:

- Delivery status and failed deliveries.
- Sender and recipient domains and servers.
- Number of deliveries and bytes per-day and per-hour.
- Recipients by domain.

- Deliveries by relays, by size, and by delay.
- Delays by server and by domain.
- The pair of correspondents that exchanged the highest number of mails.

Lire would not be a suitable base for this project: it does not extract enough data; does not deal with rejected delivery attempts in any way; does not make the extracted data easily available to other programs. Its parser is not extensible; it could easily be replaced, but that would require writing a parser from scratch, so would not be any less work than writing PLP. PLP could possibly be used to parse Postfix log files for Lire, but the difficulty may outweigh the benefits, e.g. Lire's data store may not be capable of storing the data extracted by PLP, but the lack of documentation hinders any evaluation. As with YAALA this approach has not been attempted by the author.

3.10 Logrep

> *Logrep is a secure multi-platform framework for the collection, extraction, and presentation of information from various log files. It features HTML reports, multi dimensional analysis, overview pages, SSH communication, and graphs, and supports over 30 popular systems including Snort, Squid, Postfix, Apache, Sendmail, syslog, ipchains, iptables, NT event logs, Firewall-1, wtmp, xferlog, Oracle listener and Pix.*
> `http://www.itefix.no/i2/index.php`
> Last checked 2009/01/11.

Logrep extracts fewer than half the fields PLP does:

- For mail sent and received: from address, size, and time and date. Which date and hour are stored is not documented: start time, end time, delivery time, or another time?
- For mail sent: to addresses, SMTP code, and delay.

- For mail received: the hostname of the sender.

It also counts the number of log lines parsed and skipped. It omits client IP address and hostname, server IP address, HELO hostname, queueid, timestamps of each log line, enhanced status code, and message-id. Log lines are correlated based on the queueid (called sessionname [sic] within Logrep), but not by pid. The parsing is error prone: empty fields are saved when the log line does not match the regex, though it appears that they will not overwrite existing data. Most notably, rejected delivery attempts are completely ignored, making it unsuitable for the purposes of this project. It does not try to address any of the complications in §5.7 [p. 107] except for correlating log lines by queueid.

Logrep does not come with any documentation, though some scant documentation is available on its website (PLP provides copious documentation). It requires a web browser to interact with it, so automated log file processing would be difficult, whereas enabling automated processing is a key part of PLP's design. Sadly, all the author's attempts to use Logrep failed, because it was unable to access the log files selected for parsing; this appears to be a bug rather than operator error. If the problem was caused by operator error, the interface needs improvement because the (minimal) instructions were followed as closely as possible, and multiple attempts were made. Because parsing failed it was not possible to review the reports Logrep can generate (available in HTML only), or to examine the (undocumented) format it uses to save extracted data for subsequent reuse.

Logrep extracts far less data from Postfix log files than PLP, completely ignores rejected delivery attempts, is effectively undocumented, does not deal with the more complicated aspects of Postfix log files, and does not work properly.

3.11 Summary

This chapter has reviewed ten programs that perform basic Postfix log file parsing, some to a greater level of detail than others. None of the reviewed parsers collect the breadth of information gathered by PLP, and none are designed to be extensible to handle new log lines. Some correlate log lines by queueid, but none correlate by pid; none deal with any of the other complications described in §5.7 [p. 107]. All of the reviewed parsers generate a report, and some provide a greater or lesser degree of customisation. Most have a data store, but only LMA provides any documentation on its format; some deliberately make the data store inaccessible to other tools. Most but not all of the parsers provide documentation, with the quality ranging from unusable to excellent. Fewer than half of the parsers were capable of parsing the 93 test log files; improving or extending parsing would have been quite a difficult task for any of the parsers, and one that the author did not have the time to attempt. Table 3.1 on the following page provides a summary of the parsers' features. The overriding difference between PLP and the other parsers reviewed herein is that none of them aim for the high level of understanding of Postfix log files achieved by PLP.

Table 3.1: Summary of reviewed parsers' features

Parser	Parsed test log files?	Data store?	Custom reports?	Documentation quality?	Source code?
LMA	No	Yes	No	Poor	Yes
Pflogsumm	Yes	No	Partial †	Good	Yes
Sawmill	Yes	Yes	Searches	Excellent	Yes α
Splunk	Yes	Yes	Searches & reports	Abundant but poor	No
Isoqlog	No	Yes	No	No English documentation	Yes
AWStats	Partially	Yes	Partial †	Good	Yes
Anteater	No	No	No	None	Yes
YAALA	No	Yes ‡	Searches & reports	Poor	Yes
Lire	Yes	Yes	Yes	Poor	Yes
Logrep	No	Yes	No	None	Yes
PLP	Yes	Yes β	No χ	ε	Yes

† Sections can be omitted from a report, but extra sections cannot be added.

‡ YAALA only stores the data required to produce the latest report; other data will be discarded.

α Sawmill's source code is available, but in an encrypted or obfuscated form.

β PLP is the only parser with documentation for the format of its data store.

χ PLP defers report generation to subsequent programs, but all the necessary documentation to produce reports is provided.

ε PLP aims to have thorough and complete documentation, but the author cannot provide an unbiased review.

Chapter 4

Parser Architecture

The parser architecture described in this chapter is flexible enough to be used as the basis of other parsers. Obviously it is particularly suitable for writing parsers for log files; with judicious use of cascaded parsing (see §4.3 [p. 59]), a calculator could easily be implemented; a more ambitious project might attempt to parse a programming language.[1] To avoid cluttering the description of the architecture with the details of implementing a parser for Postfix log files, each topic has been given its own chapter. This chapter is focused on the theoretical, implementation-independent aspects of the architecture; the practical difficulties of writing a parser for Postfix log files are covered in detail in §5 [p. 70]. This chapter presents the architecture developed for this project, beginning with the overall architecture and design, followed by detailed documentation of the three components of the architecture: Framework, Actions, and Rules.

4.1 Architecture Overview

It should be clear from the earlier Postfix background (§2.3 [p. 22]) that Postfix log files may vary widely from host to host. With this in mind, one of the architecture's design aims was to make parsing new inputs as effortless as possible, to enable administrators to properly parse their own log files. The

[1] Lisp, where every statement is enclosed in parentheses and can easily be isolated from the surrounding statements, might be particularly amenable.

solution developed is to divide the architecture into three parts: Framework, Actions, and Rules. Each will be documented separately, but first an overview:

Framework The framework is the structure that actions and rules plug into. It manages the parsing process, providing shared data storage, loading and validation of rules, storage of results, and other support functions.

Actions Each action performs the processing required for a single *category* of inputs, e.g. rejection of a delivery attempt. Actions are invoked to process an input once it has been recognised by a rule.

Rules Rules are responsible for classifying inputs: each rule recognises one input *variant* — a single input category may have many input variants. Each rule also specifies the action to be invoked when an input has been recognised; rules thus provide an extensible method of associating inputs with actions.

For each input, the framework tries each rule in turn until it finds a rule that recognises the input, then invokes the action specified by that rule. If the input is not recognised by any of the rules, the framework issues a warning; the framework will usually continue parsing after this, although some parsers might prefer to stop immediately.

Decoupling the parsing rules from their associated actions allows new rules to be written and tested without requiring modifications to the parser source code, significantly lowering the barrier to entry for casual users who need to parse new inputs, e.g. part-time systems administrators attempting to combat and reduce spam; it also allows companies to develop user-extensible parsers without divulging their source code. Decoupling the framework, actions, and rules simplifies all three and creates a clear separation of functionality: the framework manages the parsing process and provides services to the actions; actions benefit from having services provided by the framework, freeing them to concentrate on the task of accurately and correctly processing inputs and

the information provided by rules; rules are responsible for recognising inputs, and extracting data from those inputs for processing by actions.

Separating the rules from the actions and framework makes it possible to parse new inputs without modifying the core parsing algorithm. Adding a new rule with the action to invoke and a Regular Expression (regex) to recognise inputs is trivial in comparison to understanding an entire parser, identifying the correct location to change, and making the appropriate changes. Changes to a parser must be made without adversely affecting existing parsing, including any edge cases that are not immediately obvious; an edge case that occurs only four times in 93 log files is described in Yet More Aborted Delivery Attempts (§5.7.12 [p. 122]). The more intrusive the changes are, the more likely they are to introduce a bug, so reducing the extent of the changes is important. Requiring changes to a parser's source code also complicates upgrades of the parser, because the changes must be preserved during the upgrade, and they may clash with changes made by the developer. This architecture allows the user to add new rules to a parser without having to edit it, unless the new inputs cannot be processed by the existing actions. If the new inputs do require new functionality, new actions can be added to the parser without having to modify existing actions; only when the new actions need to cooperate with existing actions will more extensive changes be required.

Some similarity exists between this architecture and William A. Wood's Augmented Transition Networks (ATN) [12, 29], used in Computational Linguistics to create grammars that parse or generate sentences. The resemblance between the two (shown in table 4.1 on the next page) is accidental, but clearly the two different approaches share a similar division of responsibilities, despite having different semantics.

The architecture can be thought of as implementing transduction: it takes data in one form and transforms it to another form; Postfix Log Parser (PLP) transforms log files to a Structured Query Language (SQL) database.

Unlike traditional parsers such as those used when compiling a programming language, this architecture does not require a fixed grammar specification that inputs must adhere to. The architecture is capable of dealing with in-

Table 4.1: Similarities between ATN and this architecture

ATN	Architecture	Similarity
Networks	Framework	Determines the sequence of transitions or actions that constitutes a valid input.
Transitions	Actions	Assemble data and impose conditions the input must meet to be accepted as valid.
Abbreviations	Rules	Responsible for recognising inputs.

terleaved inputs, out of order inputs, and ambiguous inputs where heuristics must be applied — all have arisen and been successfully accommodated in PLP. This architecture is ideally suited to parsing inputs where the input is not fully understood or does not conform to a fixed grammar: the architecture warns about unparsed inputs and other errors, but continues parsing as best it can, allowing the developer of a new parser to decide which deficiencies are most important and the order to address them in, rather than being forced to fix the first error that arises.

4.2 Framework

The framework manages the parsing process and provides support functions for the actions, freeing the programmers writing actions to concentrate on writing productive code. It links actions and rules, allowing either to be improved independently of the other, and allows new rules to be written without needing changes to the source code of a parser. The framework is the core of the architecture and is deliberately quite simple: the rules deal with the variation in inputs, and the actions deal with the intricacies and complications encountered during parsing. Finding the rule that recognises the input is a very simple process, as shown by the pseudo-code in figure 4.1 on the following page. The framework tries each rule until it finds one that recognises the input, then it invokes the action specified by the rule. The framework issues a warning if the input is not recognised by any of the rules.

Most parsers will require the same basic functionality from the framework;

Figure 4.1: Pseudo-code showing how the framework finds the rule that recognises the input

```
INPUT:
for each input {
    for each rule defined by the user {
        if this rule recognises the input {
            invoke the action specified by the rule
            next INPUT
        }
    }
    warn the user that the input was not recognised
}
```

it is responsible for managing the parsing process from start to finish, which will generally involve the following:

Register actions Each action needs to be registered with the framework so that the framework knows about it: the list of registered actions will be used when validating rules.

Load and validate rules The framework loads the rules from wherever they are stored: a simple file, a database, or possibly even a web server or other network service — though that would have serious security implications. It validates each rule to catch problems as early in the parsing process as possible; the checks will be implementation-specific to some extent, but will generally include the following:

- Ensuring the action specified by the rule has been registered with the framework.
- Checking for conflicts in the data to be extracted, e.g. setting the same variable twice.
- Checking that the regex in the rule is valid.

Some optimisation steps may also be performed during loading of rules, as described in §6.1 [p. 130].

Convert physical inputs to logical inputs Each rule recognises a single input at a time: there is no facility for rules to consume more input or push unused input back onto the input stream, although actions may use cascaded parsing (explained in §4.3 on the following page) to push input back onto the input stream. A physical input (e.g. a single line read from the input stream) may contain multiple or partial logical inputs, and the framework must pre-process these to provide a logical input for the rules to recognise. This simplifies the rules and actions considerably, at the cost of added complexity in the framework; during the design phase it was decided that it was easier to deal with the problem of parsing multiple or partial inputs once, rather than dealing with it in every rule and action. This is trivial for Postfix log files because they have a one-to-one mapping between physical and logical inputs; mapping between physical and logical inputs may be more difficult for other types of input. Some input types may require pre-processing equivalent to parsing the physical inputs; in such cases the framework should take the approach adopted by many other parsers: combine the physical inputs into one complete input, use the rules to recognise the start of the input, discard the recognised portion if successful, and repeat until the input has been exhausted.

Classify the input The pseudo-code in figure 4.1 on the previous page shows how rules are successively tried until one is found that recognises the input. That pseudo-code is very simple: there may be efficiency concerns (§6.1 [p. 130]), rule conditions (§4.4.2 [p. 63]), or rule priorities (§4.4 [p. 61]) that complicate the process.

Invoke actions Once a rule has been found that recognises the input, the specified action will be invoked. The framework marshals the data extracted by the rule, invokes the action, and pushes the modified input onto the input stream if the action uses cascaded parsing (see §4.3 on the next page).

Shared storage Parsers commonly need to save some state information

about the input being parsed, e.g. a compiler tracking which variables are in lexical scope as it moves from one lexical block to another. The framework provides shared storage to deal with this and any other storage needs the actions may have. Actions may need to exchange data to correctly parse the input, e.g. setting or clearing flags, maintaining a list of previously used identifiers, or ensuring at a higher level that the input being parsed meets certain requirements.

Save and load state The architecture can save the contents of the shared storage it provides for actions, and reload it later so that information is not lost between parsing runs. PLP does this because mails may take some time to deliver and thus have their log lines split between log files; a compiler might store data structures it builds as it parses different files.

Specialised support functions Actions may need support or utility functions; the framework may be a good location for support functions, but if another way exists to make those functions available to all actions it may be preferable to use that way instead, maintaining a clear separation of concerns.

4.3 Actions

Each action is a separate procedure written to process a particular category of input, e.g. rejection of a delivery attempt. One input category may have many input variants; in general each action will handle one input category, with each rule recognising one input variant. It is anticipated that parsers based on this architecture will have many actions, and each action may be invoked by many rules, with the aim of having simple rules and tightly focused actions. An action may need to process different input variants in slightly different ways, but large variation in the processing performed by an action indicates the need for a new action and a new category of input; if an action becomes overly complicated it starts to turn into a monolithic parser, with too much logic contained in a single procedure.

The ability to easily add special purpose actions to deal with difficulties and new requirements that are discovered during parser development is one of the strengths of this architecture. When a new requirement arises an independent action can be written to satisfy it, instead of having to modify a single monolithic function that processes every input, with all the attendant risks of adversely affecting the existing parsing. Sometimes the new action will require the cooperation of other actions, e.g. to set or check a flag, so actions are not always self-contained, but there will still be a far lower degree of coupling and interdependency than in a monolithic parser.

During development of PLP it became apparent that in addition to the obvious variety in log lines there were many complications to overcome. Some were the result of deficiencies in Postfix's logging, and some of those deficiencies were rectified by later versions of Postfix, e.g. identifying bounce notifications (§5.7.4 [p. 112]); others were due to the vagaries of process scheduling, client behaviour, and administrative actions. All were successfully accommodated in PLP: adding new actions was enough to overcome several of the complications; others required modifications to a single existing action to work around a difficulty; the remainder were resolved by adapting existing actions to cooperate and exchange extra data (via the framework), changing their behaviour as appropriate based on that extra data. Every architecture should aim to make the easy things easy and the hard things possible; the successful implementation of PLP demonstrates that this architecture achieves that aim.

Actions may modify the input they process and return it to the framework, where it will be parsed as if read from the input stream, allowing for a simplified version of cascaded parsing [13]. This powerful facility allows several rules and actions to parse a single input, potentially simplifying both rules and actions. A simple example is to have one rule and action removing comments from inputs, so that other rules and actions do not have to handle comments at all; obviously if comment characters can be escaped or embedded in quoted strings the implementation must be careful not to remove those. For some inputs this kind of pre-processing can greatly simplify parsing, echoing the simplification provided by the framework presenting rules and actions

with logical inputs rather than physical inputs. A more complex example of cascaded parsing is evaluating simple arithmetic expressions, where sub-expressions enclosed in parentheses must be evaluated first; cascaded parsing can be used to parse and evaluate the sub-expressions, substituting the result into the original expression for subsequent re-evaluation. Actions do not need to be specially registered with the framework or be declared in a particular way to use cascaded parsing: actions that do not use cascaded parsing will return nothing, those that do will simply return a string to be parsed.

This section is quite brief, because the architecture deliberately imposes as few restrictions, conditions, and conventions as possible on actions, to allow maximum flexibility for parsers based on this architecture.

4.4 Rules

Rules are responsible for recognising inputs: each rule should recognise one and only one input variant; an input category with multiple input variants should have multiple rules, one for each variant. Rules will typically use a regex when recognising inputs, but other approaches may prove useful for some applications, e.g. comparing fixed strings to the input, or checking the length of the input; for the remainder of this thesis it will be assumed that a regex is used. Each rule must specify the regex to recognise inputs and the action to invoke when recognition is successful, but implementations are free to add any other attributes they require; §5.3.2 [p. 79] describes the attributes used in PLP, and some generally useful attributes will be discussed later in this section.

Using the rules is simple: the first rule to recognise the input determines the action that will be invoked; there is no backtracking to try alternate rules, and no attempt is made to pick a *best* rule. §4.4.2 [p. 63] contains an example which requires that the rules are used in a specific order to correctly parse the input, so a mechanism is needed to allow the author of the rules to specify that ordering. Each rule can have a priority attribute: when recognising inputs, the framework should try the rules in the order specified by their priority attributes, giving the ruleset author fine-grained control over the

order that rules are used in. The priority attribute may be implemented as a number, or as a range of values, e.g. low, medium, and high, or in a different fashion entirely if it suits the implementation. Rule ordering for efficiency is a separate topic that is covered in §6.1.2 [p. 137]; overlapping rules are discussed in §4.4.3 [p. 65].

In Context Free Grammar (CFG) terms the rules could be described as:

<input> ↦ rule-1 | rule-2 | rule-3 | ... | rule-n

This is not entirely correct because the rules are not truly context free: rule conditions (described in §4.4.2 on the next page) restrict which rules will be used to recognise each input, imposing a context of sorts.

4.4.1 Adding New Rules

The framework issues a warning for each unparsed input, so it is clearly evident when the ruleset needs to be augmented. Parsing new inputs is achieved in one of three ways:

1. Modify an existing rule's regex, because the new input is part of an existing variant.

2. Write a new rule that pairs an existing action with a new regex, adding a new variant to an existing category.

3. Create a new category of inputs, write a new action to process inputs from the new category, and write a new rule pairing the new action with a new regex.

Decoupling the rules from the actions and framework enables other rule management approaches to be used, e.g. instead of manually editing existing rules or adding new rules, machine learning techniques could be used to automate the process. If this approach was taken, the choice of machine learning technique would be constrained by the size of typical data sets (see §6.1 [p. 130]). Techniques requiring the full data set when training would be impractical; Instance Based Learning [8] techniques that automatically determine which inputs from the training set are valuable and which inputs

can be discarded might reduce the data required to a manageable size. A parser could also dynamically create new rules in response to certain inputs, e.g. parsing a subroutine declaration could cause a rule to be created that parses calls to that subroutine, checking that the arguments used agree with the subroutine's signature. These avenues of research and development have not been pursued by the author, but the architecture allows them to easily be undertaken independently.

4.4.2 Attaching Conditions to Rules

Rules can have conditions attached that will be evaluated by the framework before attempting to use a rule to recognise an input: if the condition is true the rule will be used, if not the rule will be skipped. Conditions can be as simple or complex as the parser requires, though naturally as the complexity rises so too does the difficulty in understanding how different rules and actions interact. The framework has to evaluate each condition, so as the complexity of conditions increases so will the complexity of the code required to evaluate them. Beyond a certain level of complexity, conditions should probably be written in a proper programming language, e.g. taking advantage of dynamic languages' support for evaluating code at run-time, or embedding a language like Lua into the parser. If an implementation is going to use conditions so complex that they will require a Turing-complete programming language, the design may need to be revisited, including the decision to use this architecture — there may be other architectures more suitable.

Conditions that only examine the input will be the easiest conditions to understand, because they can be understood in isolation; they do not depend on variables set by actions or other rules. Conditions that examine the input can be complex if required, but simple conditions can be quite useful too, e.g. every Postfix log line contains the name of the Postfix component that produced it, so every rule used in PLP has a condition specifying the component whose log lines it recognises, reducing the number of rules that will be used when recognising a log line (see §5.6.4 [p. 106] for details) and increasing the chance that the log line will be correctly recognised.

Conditions can also check the value of variables that have been set by either actions or rules; it is easier to understand how a variable's value will be used and changed if it is set by rules only, rather than by actions, because the chain of checking and setting variables can be followed from rule to rule. The downside to actions setting variables that are used in rule conditions is action at a distance: understanding when a rule's condition will be true or false requires understanding not just every other rule but also every action. The framework probably does not need to support a high level of complexity and flexibility when rules are setting variables; however, if the framework supports complex conditions, that code can probably be easily extended to support complex variable assignments too. The level of complexity the framework supports when evaluating conditions and setting variables has two costs that must be taken into account when designing a parser: the difficulty of implementation, and the difficulty of understanding or writing correct rules; the flexibility provided by complex conditions may be outweighed by the difficulty in understanding the interactions between rules that use them.

An example of how rule conditions can be used is parsing C-style comments, which start with `/*` and end with `*/`; the start and end tokens can be on one line, or may have many lines between them. Table 4.2 on the following page shows the regexes, conditions, and state changes of the four rules required to parse C-style comments. These are simplified rules, e.g. rules one and two will incorrectly recognise the comment start token if it is within a quoted string. Rules one and two will be used when the parser is parsing code, not comments: rule one recognises a comment that is contained within one line and leaves the parser's state unchanged; rule two recognises the start of a comment and changes the parser's state to parsing comments instead of parsing code. Rules three and four will be used when the parser is parsing a comment: rule three recognises the end of a comment and switches the parser's state back to parsing code; rule four recognises a comment line without an end token and keeps the parser's state unchanged. It is important that the rules are applied in the order listed in table 4.2 on the next page because rule two overlaps with rule one, and rule four overlaps with rule three; §4.4 [p. 61] has explained how this is achieved. §4.4.3 on the next page will discuss the

benefits and difficulties of using overlapping rules.

Table 4.2: Rules to parse C-style comments

No.	Regex	Condition	Variable Changes
1	`/\*.*?\*/`	state == parsing code	
2	`/\*.*`	state == parsing code	state = parsing comment
3	`.*\*/`	state == parsing comment	state = parsing code
4	`.*`	state == parsing comment	

Note that "state" is merely a descriptive name for the variable; the variable can be called anything at all.

4.4.3 Overlapping Rules

When adding new rules, the rule author must be aware that the new rule may overlap with one or more existing rules, i.e. some inputs could be parsed by more than one rule. Unintentionally overlapping rules lead to inconsistent parsing and data extraction because the first rule to recognise the input wins, and the order in which rules are used might change between parser invocations. Overlapping rules are frequently a requirement, allowing a more specific rule to recognise some inputs and a more general rule to recognise the remainder, e.g. separating Simple Mail Transfer Protocol (SMTP) delivery to specific sites from SMTP delivery to the rest of the world. Using overlapping rules simplifies both the general rule and the more specific rule. Overlapping rules should have a priority attribute to specify their relative ordering; negative priorities may be useful for catchall rules. The architecture does not try to detect overlapping rules: that responsibility is left to the author of the rules.

Overlapping rules can be detected by visual inspection, or a program could be written to analyse the regexes in a ruleset. Traditional regexes are equivalent in computational power to Finite Automata (FA) and can be converted to FA, so regex overlap can be detected by finding a non-empty intersection of two FA. Perl 5.10 regexes [15] are more powerful than traditional regexes: they can match correctly balanced brackets nested to an arbitrary depth, e.g. `/^[^<>]*(<(?:(?>[^<>]+)|(?1))*>)[^<>]*$/` matches `z<123<pq<>rs>j<r>ml>s`. Matching balanced brackets requires the

regex engine to maintain state on a stack, so Perl 5.10 regexes are equivalent in computational power to Push-Down Automata (PDA); detecting overlap may require calculating the intersection of two PDA instead of two FA. PDA intersection is not closed, i.e. the result cannot always be implemented using a PDA, so intersection may be intractable sometimes, e.g.: $a^*b^nc^n \cap a^nb^nc^* \rightarrow a^nb^nc^n$. Detecting overlap amongst n regexes requires calculating $\frac{n(n-1)}{2}$ intersections, resulting in $O(n^2x)$ complexity, where x is the cost of calculating FA or PDA intersection. This is certainly not a task to be performed every time a parser runs: naive detection of overlap amongst PLP's 184 rules would require calculating 16,836 intersections.

When detecting overlap, any conditions attached to rules must be taken into account, because two rules whose regexes overlap may have conditions attached that prevent the rules overlapping. A less naive approach to overlap detection would first check for overlapping conditions amongst rules, and then check for overlap between the regexes of each pair of rules with overlapping conditions. Rule overlap is not transitive, e.g. given these three conditions:

1. total < 10
2. total > 20
3. total < 30

The first and second conditions do not overlap, but the third condition overlaps with both the first and second conditions. When rules are paired based on how their conditions overlap, the complexity of detecting overlap amongst n rules is $O(n^2y + |o|x)$, where:

y = cost of checking for overlap between two conditions
o = set of pairs of rules with overlapping conditions
x = cost of checking for overlap between two regexes

In the worst case, $|o|$ above will be equal to n^2. For this approach to be more efficient than the naive approach, y must be significantly lower than x. If y is higher than x then the checks for overlap should be performed in the opposite order: check for regex overlap first, then check for condition overlap only between pairs of rules with overlapping regexes.

Once conditions pass a certain level of complexity, determining if two conditions overlap becomes intractable, because it requires so much knowledge of other rules and possibly even actions. For example, given two rules with conditions `verbose == true` and `silent == true`, logically these rules should not overlap, yet there is nothing to stop both variables being set to true by one or more rules or actions. If variables used in conditions can be set by actions, determining if two conditions overlap is impossible: the Halting Problem shows that it is impossible for one program to answer questions about another program's behaviour if the inquiring program is implemented on a computational machine whose power is equal or less than the power of the computational machine the other program is implemented upon.

4.4.4 Pathological Rules

It is possible to write pathological regexes, which fall into two main categories: regexes that match inputs they should not, and regexes that consume excessive amounts of CPU time during matching. Defining a regex that matches inputs it should not is trivial: `/^/` matches the start of every input. This regex would be found by a tool that detects overlapping rules, and would easily be noticed by visual inspection, but more complex regexes would be harder to find. Regexes that match inputs more than they should are a problem not because of excessive resource usage, but because they may prevent the correct rule from recognising the input. If an adaptive ordering system is used to prioritise rules that frequently recognise inputs (see §6.1.2 [p. 137]), then a rule with a regex that matches inputs it should not may be promoted up through the list, displacing an increasing number of correct rules as it rises.

Excessive CPU time is usually consumed when a regex fails to match an input, and the regex engine backtracks many times because of alteration or nested quantifiers; successful matching is generally quite fast with such regexes, so problematic regexes may go unnoticed for a long time. For example, with most regex engines matching double quoted strings with `\"([^"\\]+|\\.)*"\` is very fast when a match can be found, but when the match fails its computational complexity for a string of length n is

$O(2^n)$; see [6] for in-depth discussion of nested quantifiers, backtracking, alteration, and capturing groups. Pathological regexes that consume excessive CPU time can be difficult to detect, whether by visual inspection or by machine inspection, but if a regex is converted to a FA or the internal representation used by the regex engine, it may be possible to determine if nested quantifiers or other troublesome constructs are present. Modern regex engines have addressed many of these problems, e.g. the regex to match double quoted strings given above fails immediately with Perl 5.10, regardless of the input length, because the regex engine looks for both of the required double quotes first. Similarly, Perl 5.10's regex engine optimises alterations starting with literal text into a trie, which has matching time proportional to the length of the alternatives, rather than the number of alternatives. Perl regexes can use `(?>pattern)`, which matches `pattern` the first time the regex engine passes over it, and does not change what it originally matched if the regex engine backtracks over it, alleviating problems caused by excessive backtracking; Prolog users will notice a similarity to the `!` (cut) operator. A presentation showing some of Perl 5.10's new regex features is available at `http://www.regex-engineer.org/slides/perl510_regex.html` (last checked 2009/03/03).

Conditions can vary in complexity from simple equality through to a Turing-complete language, so enumerating pathological conditions is difficult if not pointless. Conditions that check variables or the input in uncomplicated ways may exhibit unexpected or incorrect behaviour, but are unlikely to exhibit pathological behaviour. Deciding if a more complex condition's behaviour is pathological or simply a bug is difficult and to some extent is a matter of opinion. When this architecture has received more widespread usage, consensus should be reached on the topic of pathological conditions.

4.5 Summary

This chapter has presented the parser architecture developed for this project. It started with a high level view of the architecture, describing how it achieves its design aim of being easily extensible for users, and the advantages that

being easily extensible brings to parser authors. The three main components of the architecture were documented in detail, explaining each component's responsibilities and the functionality it provides, plus any difficulties associated with the components. The framework provides several support functions and manages the parsing process, enabling simple rules and actions to be written. The actions are simple to understand, because the architecture does not impose any structure or requirements upon them: parser authors are free to do anything they want within an action. The architecture's support for cascaded parsing was described in the actions section, with an example to illustrate how it can be useful for general parsing. The rules section was the longest section in this chapter, because although the rules appear to be quite simple — recognise an input and specify the action to invoke — they have subtle behaviour that needs to be clearly explained. When extending a ruleset, a decision needs to be taken about whether the input should be recognised by extending an existing rule, by adding a new rule to an existing input category, or by adding a new input category, action, and rule. Rules can have conditions attached to them, restricting the set of rules used to recognise an input; the complexity of the conditions used greatly influences the difficulty of writing a correct ruleset or understanding and extending an existing ruleset. Overlapping rules are frequently a requirement in a parser, and their use can greatly simplify some rules, but they can be a source of bugs because they can recognise inputs unexpectedly. The framework does not try to detect overlapping rules, because overlap amongst rules may be valid and is quite often intentional; that responsibility falls to the author of the ruleset. The difficulty involved in detecting overlap is proportional to the complexity of a ruleset's regexes and conditions, and may be possible for a human yet intractable or impossible for a program. The rules section concludes with a discussion of pathological rules, concentrating on pathological regexes.

Chapter 5

Postfix Log Parser

This chapter documents the implementation of Postfix Log Parser (PLP), a parser for Postfix log files based on the architecture described in §4 [p. 53]. Any design may look good when examined in the abstract, but the real test of the design comes with the first concrete implementation; only then do any practical difficulties come to light. Implementing this architecture was straightforward — the difficulties came not from the architecture, but from anomalies in the log files and Postfix's behaviour, as described in §5.7 [p. 107]. PLP successfully deals with all the difficulties that were discovered during its development, but further difficulties may arise during future usage; unfortunately solving all potential difficulties that may arise is an impossible task, but the descriptions of the solutions developed thus far should help if someone needs to solve a problem of their own.

PLP deals with all the eccentricities and oddities of parsing Postfix log files, presenting the resulting data in a normalised, simple to use representation. Unfortunately, dealing with the complications that arise sometimes requires the parser to discard log lines (e.g. §5.7.5 [p. 113]), and, less frequently, to discard a data structure (e.g. §5.7.7 [p. 116]). PLP can parse log files from Postfix version 2.2 through to version 2.5, and should parse log files from later versions with only minor modifications or an updated ruleset. The parser's task is to follow the journey each mail takes through Postfix, combining the data captured by rules into a coherent whole, and saving it in a useful

and consistent form. The intermingling of log lines from different mails immediately rules out the possibility of handling each mail in isolation; the parser must handle multiple mails in parallel, each potentially at a different stage in its journey, without any interference between mails — except in the minority of cases where intra-mail interference is required, e.g. mail re-injected for forwarding (§5.7.3 [p. 110]). The best way to deal with intermingling of log lines is to maintain state information for every unfinished mail, and to manipulate the appropriate mail correctly for each log line encountered.

This chapter begins with the assumptions under which PLP was designed and written, followed by a flowchart showing the most common paths taken through Postfix and PLP, with a description of the stages and stage transitions. The second topic is the Structured Query Language (SQL) database that provides storage for the parser: any future work that analyses gathered data will do so using the database, so the database schema acts as an Application Programming Interface (API). A diagram of the database schema is provided, plus documentation for every table and field.

The next three sections document the implementation of the three components of the architecture. First is the framework, including its initialisation phase, the parsing process, and the conveniences it offers to users of PLP; that section finishes by describing features that are important for this thesis but not strictly necessary for parsing: the performance data collected by the framework, the optimisations that can be disabled to show their effect, and the debugging options the framework provides. The second component of the architecture is the actions, starting with a graph showing how often each action is specified by rules, a description of why some actions are more popular than others, and how this popularity has no influence on how often an action is invoked. Every action that is part of PLP is documented, and the actions section concludes with a description of the process of adding a new action. Rules are the third component of the architecture, and also the most visible to advanced users, e.g. systems administrators, because it is likely that they will need to add their own rules to recognise their own log lines. A sample rule used by PLP is examined, with every field clearly documented; that is followed by a description of adding new rules, and how to determine

the value of each of the new rule's fields. PLP provides a utility to create Regular Expression (regex)es from unrecognised log lines; the algorithm it uses is documented, including the differences between it and the original algorithm it is based on. The rules section finishes with a discussion of how PLP uses rule conditions and overlapping rules, plus a description of the regex snippets provided to ease the process of writing regexes.

On first inspection, Postfix log files look like they will be simple to parse, but this impression turns out to be incorrect. The many complications and difficulties encountered while writing PLP, and the solutions developed to overcome them, are documented in §5.7 [p. 107]. This chapter finishes with a list of PLP's limitations, and some possible improvements that could be implemented.

5.1 Assumptions

PLP makes a few (hopefully safe and reasonable) assumptions:

- The log files are whole and complete: nothing has been removed, either deliberately or accidentally (e.g. log file rotation gone awry, filesystem filling up, logging system unable to cope with the volume of log messages). On a well run mail server it is extremely unlikely that any of these problems will arise, though the likelihood increases when suffering from a deluge of spam or a mail loop. When parsing individual log files in isolation, it is highly likely that some mails will have log lines in previous log files, and others will have log lines in subsequent log files; to alleviate this problem PLP supports saving and loading its state tables, so they can be preserved between log files.

- Postfix logs enough information to make it possible to accurately reconstruct the actions it has taken. Heuristics are used in several places when parsing; see §5.7.4 [p. 112], §5.7.5 [p. 113], and §5.7.9 [p. 119] for details. At least one difficulty encountered while writing PLP (§5.7.12 [p. 122]) could not be solved using the data in the log files, and requires a brute force approach.

- The Postfix queue has not been tampered with, causing unexplained appearance or disappearance of mail. This may happen if the administrator deletes mail from the queue without using `postfix/postsuper`, or if the server suffers from filesystem corruption.

In some ways this task is similar to reverse engineering or replicating a black box system based solely on its inputs and outputs. Thus far, analysis of the log files has been enough to reconstruct Postfix's behaviour, but for other programs the techniques described in [7] may be useful. Some advantages come from treating Postfix as a black box during parser development:

- Reading and understanding the source code would require a significant investment of time: Postfix 2.5.5 has 17MB of source code. Each subsequent version would require further work to investigate the changes; many of those changes, although they improve Postfix's internals, would not have any effect on its externally observable behaviour.

- PLP is developed using real world log files rather than the idealised log files someone would naturally envisage when reading the source code, which cannot accurately communicate the variety of orderings in which log lines are found in log files.

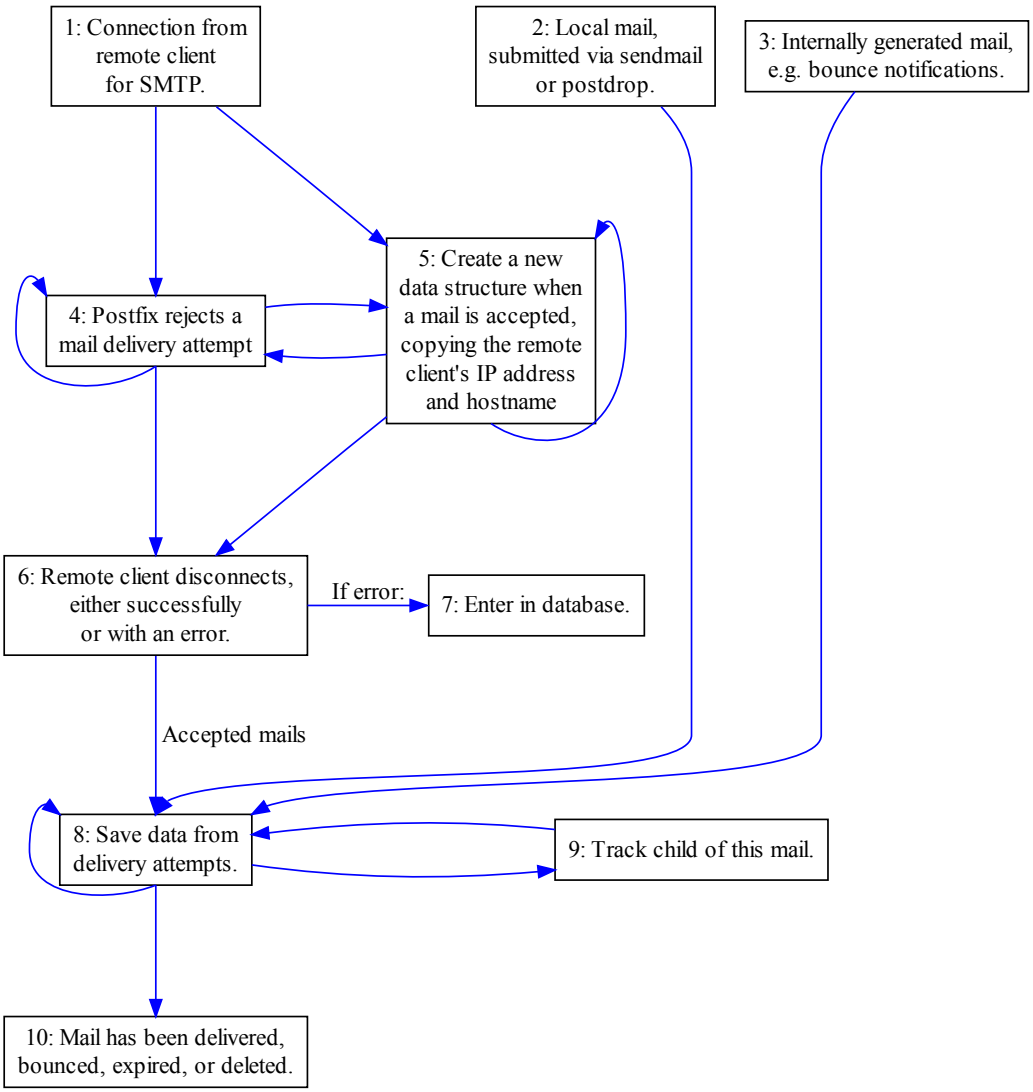
- The parser acts as a second source of information about Postfix's operation, based on empirical evidence gathered from log files. A separate project could compare the empirical knowledge inherent in PLP with Postfix's documentation and source code to see how closely the two agree.

5.2 Parser Flow Chart

The flow chart in figure 5.1 on the next page shows the most common paths a connection or mail can take through PLP; decision boxes and the difficulties described in §5.7 [p. 107] are excluded for the sake of clarity. The flow chart is intended to be a graphical overview of how a mail progresses through

both Postfix and PLP, providing an overall context into which the detailed descriptions in the remainder of this chapter will fit, in particular the actions (§5.5 [p. 89]) and complications (§5.7 [p. 107]). The states and state transitions shown in the flow chart will be explained in this section.

Figure 5.1: Postfix Log Parser flow chart



Everything starts off with a mail entering the system, whether by local submission via `postfix/sendmail` or `postfix/postdrop`, by Simple Mail Transfer Protocol (SMTP), by re-injection because of forwarding, or generated internally by Postfix. Local submission is the simplest of the four: a queueid

is assigned immediately (action: `PICKUP`; flowchart: 2), and the mail moves on to the delivery stage. Re-injection because of forwarding lacks explicit log lines of its own; it is explained fully in §5.7.3 [p. 110]. Internally generated mails lack any explicit origin in Postfix 2.2.x and must be detected using heuristics as described in §5.7.4 [p. 112]; later versions of Postfix do provide log lines for internally generated mails (action: `BOUNCE_CREATED`; flowchart: 3). Bounce notifications are the primary example of internally generated mails, though other types exist, e.g. Postfix may generate mails to the administrator when it encounters configuration errors.

SMTP is more complicated than the others:

1. The remote client connects (action: `CONNECT`; flowchart: 1).

2. This is followed by rejection of one or more mail delivery attempts (action: `DELIVERY_REJECTED`; flowchart: 4); acceptance of one or more mails (action: `CLONE`; flowchart: 5); failure of the remote client's connection (action: `DELIVERY_ERROR`; flowchart: 6); or some random interleaving of two or more of the above.

3. The client disconnects (action: `DISCONNECT` or `TIMEOUT`; flowchart: 6), either normally or with an error. If Postfix has rejected any mail delivery attempts the data gathered from those rejections will be saved to the database (action: `DISCONNECT`; flowchart: 7); if there were no rejections there will not be any data to save. Any accepted mails will already have a separate data structure, and will be delivered in the same way as mails that entered the system by any other route.

The obvious counterpart to mail entering the system is mail leaving the system, whether by deletion, bouncing, expiry, local delivery, or remote delivery. All five are handled in the same way:

1. The mail will have one or more delivery attempts (action: `MAIL_SENT` or `SAVE_DATA`; flowchart: 8).

2. Sometimes mail is re-injected for forwarding and the child mail needs to be tracked with the parent mail (action: `TRACK`; flowchart: 9); the handling of re-injected mails is described in §5.7.3 [p. 110].

3. After one or more delivery attempts the mail will be delivered (action: `MAIL_SENT`; flowchart: 8), bounced (action: `MAIL_BOUNCED`; flowchart: 8), expired (action: `EXPIRY`; flowchart: 8), or deleted by the administrator (action: `DELETE`; flowchart: 8).

4. The mail is removed from the Postfix queue. This is the last log line for this particular mail, though it may be indirectly referred to if it was re-injected. The mail is cleaned up and entered in the database, then deleted from the state tables (action: `COMMIT`; flowchart: 10).

It should be emphasised that the sequence above happens whether the mail is delivered to a mailbox, piped to a command, delivered to a remote server, bounced (because of a mail loop, delivery failure, or five day timeout), or deleted by the administrator, *unless* the mail is either parent or child of re-injection, as explained in §5.7.3 [p. 110].

5.3 Database

An SQL database is used to store both the rules and the data gleaned by parsing Postfix log files. Understanding the database schema is helpful in understanding the actions of the parser, and essential to developing further applications that utilise the data; §5.3.1 on the next page describes how the database schema functions as an API.

The database schema can be conceptually divided in two: the rules used to recognise log lines, and the data saved from the parsing of log files. Each rule has a regex to recognise log lines and capture data from them, and specifies the action to be invoked when a log line is recognised; they also have several other fields used by the parser, and several fields that aid the user in understanding the meaning of the log lines recognised by each rule. The rules are described in detail in §5.6 [p. 97], but the rules table is documented in §5.3.2 [p. 79] with the rest of the database schema.

The data saved from parsing log files is divided into two tables: connections and results. The connections table contains an entry for every mail accepted and every connection that rejected a delivery attempt; the individual fields

will be described in §5.3.3 [p. 82]. The results table will have at least one entry for each entry in the connections table; its fields will be covered in detail in §5.3.4 [p. 83]. A diagram of the database schema is provided in figure 5.2 [p. 80], to complement the in-depth descriptions of each table.

An important but easily overlooked benefit of storing the rules in the database is the link between rules and results: if more information is required when examining a result, the rule that produced the result is available for inspection because each result references the rule that created it. No ambiguity is possible about which rule or action created a particular result, eliminating one potential source of confusion.

A clear, comprehensible schema is essential when using the data extracted from log lines; it is more important when using the data than when storing it, because storing the data is a once-off operation, whereas utilising the data requires frequent searching, sorting, and manipulation of the data.

5.3.1 Using a Database to Provide an Application Programming Interface

The database populated by PLP provides a simple interface to Postfix log files. Although the interface is a database schema rather than a set of functions in a library, it provides the same benefits as any other API: a stable interface between the user and the creator of the data, allowing code on either side of the interface to be changed without adverse effects, as long as the interface is adhered to. Programs that use the database can range from the simple examples in §2.1 [p. 18] to far more complex data mining tools and machine learning algorithms.

Using a database simplifies writing programs that need to interact with the data in several ways:

1. Most programming languages have facilities for database access, allowing a developer to write programs that use the gathered data in their preferred programming language, rather than being restricted to the language the parser is written in.

2. Databases provide complex querying and sorting functionality for the user without requiring large amounts of programming. All databases have one or more programs, of varying complexity and sophistication, that can be used for ad hoc queries with minimal investment of time.

3. Databases are easily extensible, e.g.:

   - New columns can be added to tables, using DEFAULT clauses or TRIGGERS to populate them.
   - A VIEW gives a custom arrangement of data with minimal effort.
   - Triggers can be written to perform actions when certain events occur. In pseudo-SQL:

     ```
     CREATE TRIGGER ON INSERT INTO results
         WHERE sender = "boss@example.com"
             AND rule_id = rules.id
             AND rules.action = "DELIVERY_REJECTED"
         SEND PANIC EMAIL TO "postmaster@example.com";
     ```

   - Other tables can be added to the database, e.g. to cache historical or computed data, or to incorporate data from other sources.

4. Some databases support granting access on a fine-grained basis, e.g. allowing the finance department to produce invoices, the helpdesk to run limited queries as part of dealing with support calls, and the administrators to have full access to the data.

5. SQL is reasonably standard and many people will already be familiar with it; for those unfamiliar with SQL, lots of resources are available from which to learn, e.g. `http://philip.greenspun.com/sql/` (last checked 2009/02/23). Although every vendor implements a different dialect of SQL, the basics are the same everywhere. Depending on the database in use there may be tools available that reduce or remove the requirement to know SQL.

Storing the results in a database will also increase the efficiency of using those results, because the log files only need to be parsed once rather than each time the data is used; indeed the database may be used by someone with no access to the original log files.

5.3.2 Rules Table

Rules are discussed in detail in §5.6 [p. 97], but the structure of the rules table is documented here alongside the other tables in the database. Rules are created by the user, and will not be modified by PLP, except when it updates the hits and hits_total fields. Rules recognise individual log lines, capturing data to be saved in the connections and results tables, and specifying the action to invoke for each recognised log line.

Each rule must provide values for most of the fields below; all fields are required unless otherwise stated in the description.

id A unique identifier that other tables can use when referring to a specific rule. This will be assigned by the database.

name A short name for the rule.

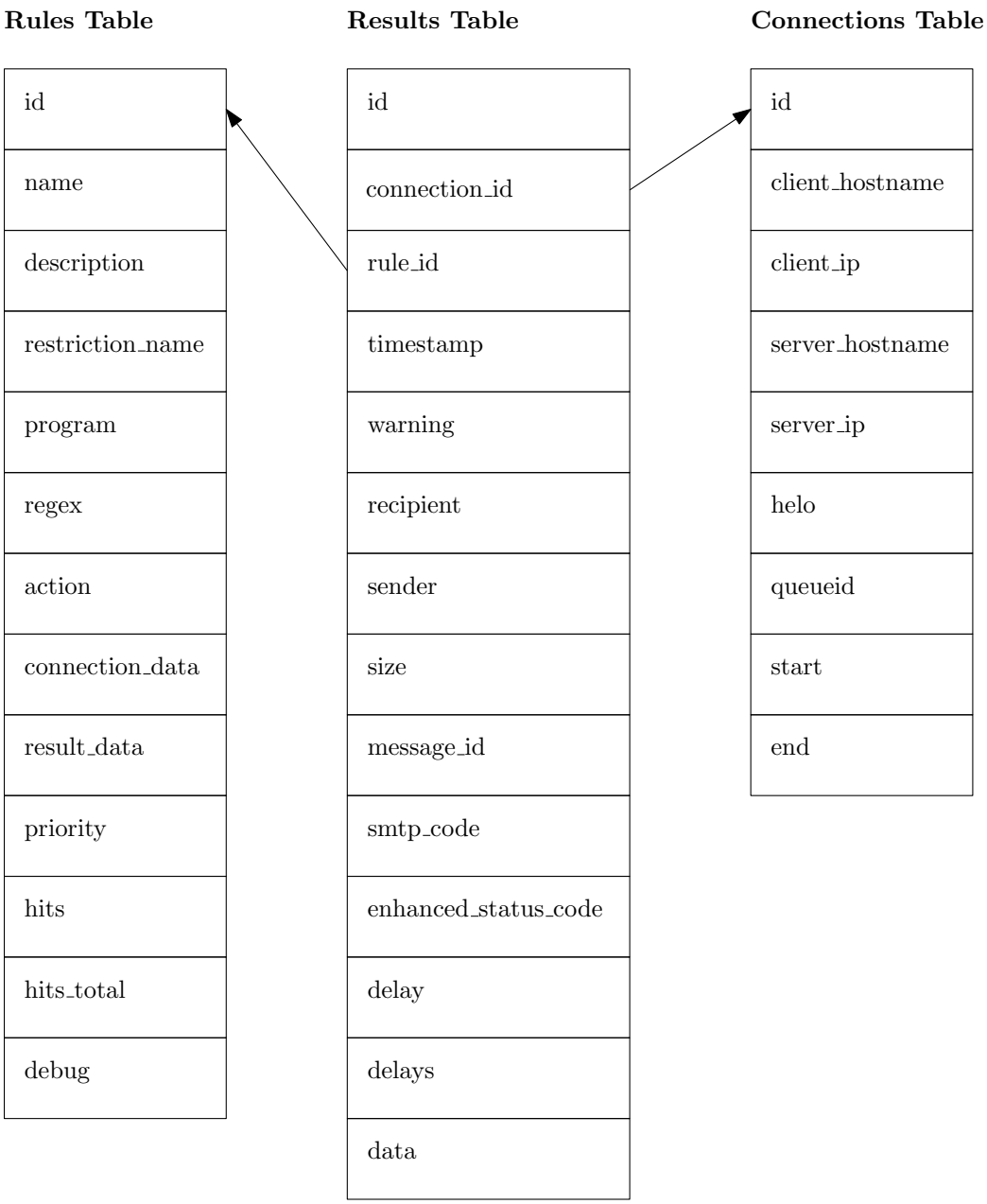
description This field should describe the event causing the log lines this rule recognises, e.g. "Mail has been delivered to the LDA (typically procmail)".

restriction_name The name of the Postfix restriction that caused the rejection of the mail delivery attempt. This field is valid only for rules that recognise rejection log lines, i.e. rules that have an action of `DELIVERY_REJECTED`.

program The Postfix component (e.g. `postfix/smtpd`) whose log lines the rule recognises; see §5.6.4 [p. 106] for full details of how this attribute is used.

regex The regex to recognise log lines with, as documented in §5.6.3 [p. 104].

Figure 5.2: Diagram of the database schema



connection_data Sometimes rules need to provide data that is not present in the log line, e.g. setting `client_ip` when a mail is being delivered to another server; any field in the connections table can be set in this way. The format is:

```
client_hostname = localhost,
  client_ip = 127.0.0.1
```

i.e. semi-colon or comma separated assignment statements. Commas and semi-colons cannot be escaped and thus cannot be included in data, because this feature is intended for use with small amounts of data and dealing with escape sequences was deemed unnecessary. This field is optional.

result_data The result table equivalent of `connection_data`, also optional.

action The action to be invoked when this rule recognises a log line; a full list of actions and the parameters they are invoked with can be found in §5.5.1 [p. 92].

hits This counter is maintained for every rule and incremented each time the rule successfully recognises a log line. At the start of each run PLP sorts the rules by hits, and at the end of the run it updates every rule's hits field in the database. Assuming that the distribution of log lines is reasonably consistent across log files, ordering rules by their recognition frequency will reduce the parser's execution time. Rule ordering for efficiency is discussed in §6.1.2 [p. 137]. This field will be set by the parser rather than the rule author.

hits_total The total number of log lines recognised by this rule over all runs of the parser; hits starts from zero each time the parser is run, but hits_total is not. This field will be set by the parser rather than the rule author.

priority This is the user-configurable companion to hits: when the list of rules is sorted by the parser, priority overrides hits. This allows more specific rules to take precedence over more general rules, as described in §4.4 [p. 61]. This field is optional.

debug If this field is true, a warning will be issued with information about the rule and the log line every time this rule recognises a log line. This field is optional.

5.3.3 Connections Table

Every accepted mail and every connection that rejected a mail delivery attempt will have a single entry in the connections table containing all of the fields below. For an incoming connection, the client is the remote machine, and the server is the local machine; for outbound mail delivery attempts, the roles are reversed.

id This field uniquely identifies the row. This will be assigned by the database.

server_ip The IP address of the server.

server_hostname The hostname of the server, it will be `unknown` if the IP address could not be resolved to a hostname via DNS.

client_ip The client IP address.

client_hostname The hostname of the client, it will be `unknown` if the IP address could not be resolved to a hostname via DNS.

helo The hostname used in the HELO command. The HELO hostname occasionally changes during a connection, presumably because spam or virus senders think it is a good idea. Postfix only logs the HELO hostname when it rejects a mail delivery attempt, but it is quite simple to rectify this as described in §5.8 [p. 126].

queueid The queueid of the mail if the connection represents an accepted mail, or `NOQUEUE` if not.

start The timestamp of the first log line.

end The timestamp of the last log line.

5.3.4 Results Table

Every recognised log line will have a row in the results table, and each row is associated with a single connection; a single connection will have at least one result associated with it, but will usually have several, and may have hundreds.

connection_id The id of the row in the connections table this result is associated with.

rule_id The id of the rule in the rules table that recognised the log line.

id A unique identifier for this result. This will be assigned by the database.

warning Administrators can configure Postfix to log a warning instead of enforcing a restriction that would reject a mail delivery attempt — a mechanism that is quite useful for testing new restrictions. This field will be false for a real rejection, or true if the log line was a warning. This field should be ignored if the result is not a rejection, i.e. the action field of the associated rule is not `DELIVERY_REJECTED`.

smtp_code The SMTP code associated with the log line. In general an SMTP code is only present in a log line representing a mail being delivered or a mail delivery attempt being rejected; results whose log

line did not contain an SMTP code will duplicate the SMTP code of other results in that connection. Some mail delivery log lines do not contain an SMTP code, e.g. when Postfix delivers to a user's mailbox; in those cases the SMTP code is specified by the rule's `result_data` field, based on the success or failure represented by the log line.

enhanced_status_code The enhanced status code [26] is similar to the SMTP code, but is intended to be interpreted by mail clients so that error messages can be clearly conveyed to the user. Enhanced status code support was added to Postfix in version 2.3; log lines from previous versions will not contain any enhanced status codes.

sender The sender's email address. This may change during a connection if the client uses different sender addresses for multiple rejected delivery attempts; however, the results for one accepted mail will only have one sender address.

recipient The recipient address; there may be multiple recipient addresses per mail or connection.

size The size of the mail, only available for delivered mails.

delay How long the mail was delayed while it was being delivered. This will only be present for delivered mails.

delays More detailed information about how long the mail was delayed while it was being delivered, again only present for delivered mails.

message_id	The message-id of the accepted mail, again it is present for delivered mails only.
data	A field available to store a piece of captured data that does not have its own specific field, e.g. the rejection message from a DNS Blacklist (DNSBL).
timestamp	The log line's timestamp.

5.4 Framework

The role of the framework in this architecture was described in detail in §4.2 [p. 56]; this section is concerned with the implementation of the framework within PLP. The framework manages the parsing process, taking care of the drudgery and boring tasks, provides services to the actions, and implements some optimisations.

The framework performs some initialisation tasks each time the parser is run, setting up several state tables that will later be used by actions. Data about the connections and mails being processed is held in `connections` and `queueids` respectively. The other state tables are used when solving complications that arise during parsing: `timeout_queueids` is used when dealing with connections that time out during the DATA phase (§5.7.7 [p. 116]); `bounce_queueids` is part of the solution to bounce notification mails being delivered before their creation is logged (§5.7.14 [p. 124]); and `postsuper_deleted_queueids` caches information about mails that were recently deleted by the administrator, so that subsequent log lines processed by the `SAVE_DATA` action can be discarded (§5.7.13 [p. 123]).

The framework verifies each of the rules when it loads the ruleset, checking:

- That the specified action is registered with the framework.
- That the regex is valid. The regex will have regex components expanded (see §5.6.3 [p. 104]), and will also be compiled for efficiency (§6.1.5 [p. 145]).

- For overlap between the data captured by the regex and additional data specified in either `connection_data` or `result_data`.
- That `connection_data`, `result_data`, and the regex captures specify valid fields to save data to.

State tables from a previous PLP run, if any, will be loaded now; the framework supports saving state at any time, without adversely affecting the parsing process. The need to track re-injected mail (§5.7.3 [p. 110]) complicates the process of saving state, because the relationships between mails must be maintained; loading state is also complicated by dealing with aborted delivery attempts (§5.7.5 [p. 113]), because a separate set of relationships between connections and mails must be re-created when the state tables are restored. The last step in loading the ruleset is to sort the rules as described in §6.1.2 [p. 137].

The framework is now ready to begin parsing. For each log line it will use those rules whose `program` field equals the program in the log line (described in §5.6.4 [p. 106]), falling back to generic rules if necessary, and finally warning if the log line is unrecognised. The repetitive nature of log files gives them high compression ratios; the framework uses a Perl module named `IO::Uncompress::AnyUncompress` to read compressed log files, saving users the trouble of uncompressing them to a temporary file before parsing begins. When used interactively, the framework displays a progress bar to show how far parsing has progressed through the log file and how long the remainder of the parsing process is likely to take. The progress bar is not as accurate when parsing compressed log files, because each compressed block uncompresses to a variable number of log lines; variation between the recognition and processing time of individual log lines also affects its accuracy, but overall the progress bar is a useful addition. PLP is primarily intended for parsing complete log files, but with minor modifications it could be used to parse a live log file, periodically checking it for new log lines; this could be very useful for programs that work best with up to date data, e.g. a program for monitoring the health of the mail system, or graphs showing activity over the last five minutes.

The framework collects data used when evaluating PLP's efficiency for the evaluation chapter (§6 [p. 129]); some of the techniques used to improve parsing speed can be turned off or altered to measure the effect they have. The data gathered with each optimisation disabled will be compared to the data gathered with it enabled, to quantify the benefit each optimisation provides. Three sets of data are collected:

1. How long it takes to parse each log file (graph 6.2 [p. 134]).
2. The number of rules used when recognising log lines (graph 6.6 [p. 136]); the framework may need to try multiple rules for each log line before it finds one that recognises it.
3. The number of log lines and rules used per log line for each Postfix component (graph 6.7 [p. 136]).

The framework has five ways that it can adapt its behaviour to demonstrate how effective each optimisation is:

1. The rule ordering used can be changed from optimal (the default, most efficient) to shuffled (intended to represent an unordered ruleset) or reverse (reverse of optimal, least efficient). The results of parsing with the three different orderings are shown in §6.1.2 [p. 137].
2. The framework can record which rule recognised each log line, and then on a subsequent run consult that information so that it uses the correct rule for each log line. This gives the best possible running time, because only one rule is used to recognise each log line, as discussed in §6.1.3 [p. 140].
3. Each regex is compiled and the result cached when the ruleset is loaded; this optimisation can be disabled and the regex compiled every time it is used when recognising log lines. The effect this optimisation has is addressed shown in §6.1.5 [p. 145].

4. Normally, when a log line has been recognised the framework invokes the action specified by the rule. Invoking the action can be skipped if desired, so the timing information shows how long is spent recognising log lines only; this time can be subtracted from the time taken by a normal run to show how long is spent processing log lines. This data is analysed in §6.1.6 [p. 147].

5. The framework can skip inserting data into the database, because that dominates the execution time of the parser; the evaluation chapter measures the speed of PLP, not the speed of the database or the disks it resides on. The effect on parsing time of storing results in the database is described at the beginning of §6.1 [p. 130].

The framework also provides several debugging options, to aid in writing or correcting rules, or figuring out why PLP is not behaving as expected. In increasing order of severity they are:

1. Individual rules can set their debug field to true, and debugging information will be printed each time they recognise a log line.

2. Each result can be extended with extra debugging information, which is useful when a warning dumps a data structure for inspection. The extra information added is: the log line's timestamp in human readable form; the entire log line; the name of the log file and the line number of the log line within it. This extra information is not stored in the database.

3. Every time a log line is recognised, the recognising rule's regex and the log line can be printed, so that the user can verify that the correct rule recognised each log line. This is equivalent to setting every rule's debug field to true.

4. Every connection and result added to the database can be dumped in a human readable form. This will result in a huge amount of debugging information, so it is only useful for small log file snippets, because otherwise the amount of information is overwhelming.

A user, typically a mail administrator, would use these options when having difficulty extending the ruleset to recognise log lines not recognised by PLP's 184 rules. Option 1 is useful for debugging a single rule, to check if it is recognising log lines it should not; figuring out that a rule is not recognising log lines that it should recognise is a more difficult task. Option 2 is useful when PLP is warning about parsing problems, because extra information about a mail will be present in the warning message. Option 3 is less useful because of the volume of data it produces, though for small log file snippets, e.g. 1000 log lines, it is possible to manually verify that the correct rule recognised each log line. Option 4 is not useful for most users: it is for debugging problems within PLP, and sufficient safeguards are in place that there should be plenty of warning messages explaining what is wrong without using this option.

5.5 Actions

Actions are the component of this architecture responsible for processing all of the inputs recognised by rules; in PLP they reconstruct the journey each mail takes through Postfix, dealing with all the complications and difficulties that arise. PLP has 23 actions and 184 rules, with an uneven distribution of rules to actions as shown in graph 5.3 [p. 91]. Unsurprisingly, the action with the most associated rules is `DELIVERY_REJECTED`, the action that processes Postfix rejecting a mail delivery attempt; next is `SAVE_DATA`, the action that saves useful information without doing any other processing. The third most common action is, perhaps surprisingly, `UNINTERESTING`: this action does nothing when invoked, allowing uninteresting log lines to be parsed without any effects; it does not imply that the input is ungrammatical or unparsed. Generally, rules specifying the `UNINTERESTING` action recognise log lines that are not associated with a specific mail, e.g. notices about configuration files changing; these log lines are recognised and processed so that the framework can warn about unrecognised log lines, informing the user that they need to augment the ruleset. Most of the remaining actions have only one or two associated rules, because that input category will only ever have one or two

log line variants, e.g. all log lines showing that a remote client has connected are recognised by a single rule and processed by the `CONNECT` action.

No correlation exists between how often an action is specified by rules and how often an action is invoked because a rule recognises a log line. Graph 5.4 on the next page shows the number of times each action was invoked when parsing the 93 log files used to generate the statistics in §6.1 [p. 130], *excluding* log files 22 & 62–68, because their contents are extremely skewed by two mail loops, described in §6.1.1 [p. 132]. The action most commonly specified by rules, `DELIVERY_REJECTED`, is the third most commonly invoked action; the most commonly invoked actions, `CONNECT` and `DISCONNECT`, are each specified by only one rule. The `SAVE_DATA` and `UNINTERESTING` actions, the second and third most commonly specified actions respectively, are halfway down the graph.

As should be expected, some actions have been invoked almost exactly the same number of times: almost every `CONNECT` will have a `DISCONNECT`, with only a 0.00042% difference between the number of times the two were invoked; every mail that enters Postfix's queue will be managed by `postfix/qmgr` (`MAIL_QUEUED`) and processed by `postfix/cleanup` (`CLEANUP_PROCESSING`), and again the two have only a 0.01648% difference between their number of invocations. `CLEANUP_PROCESSING` is invoked slightly more often than `MAIL_QUEUED`: occasionally, an accepted mail in the process of being transferred is interrupted by a timeout, and there is a log line from `postfix/cleanup` but not from `postfix/qmgr`, as described in §5.7.8 [p. 118].

Figure 5.3: Number of rules specifying each action

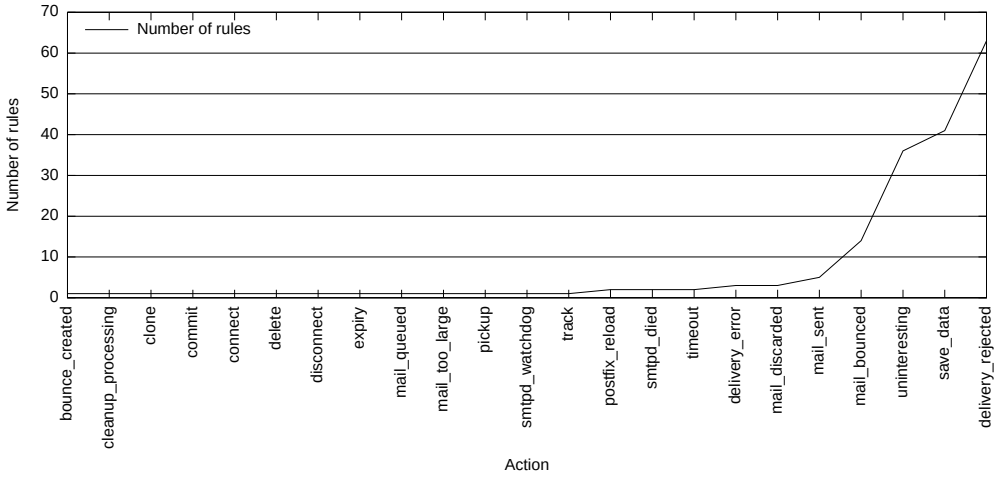
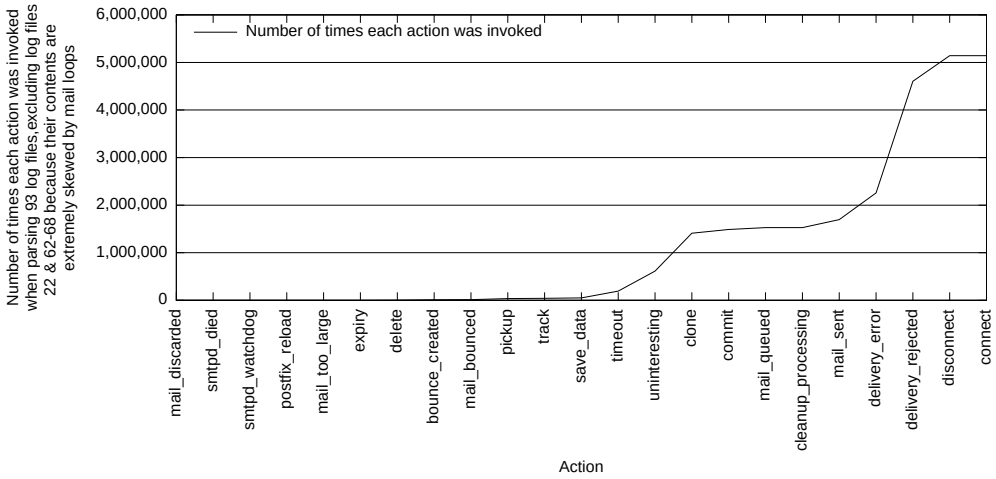


Figure 5.4: Number of times each action was invoked when parsing 93 log files, excluding log files 22 & 62–68 because their contents are extremely skewed by mail loops



5.5.1 Description of Each Action

This section documents the actions found in PLP; it may help to revisit the flow chart in §5.2 [p. 73] to see how a mail passes from one action to another as its log lines are recognised. The words *mail* and *connection* are used in the action descriptions below because they are less unwieldy and more specific than *state table entry*; a connection becomes a mail during the `CLONE` action, which processes Postfix accepting a delivery attempt, and the data structure is copied from the connections state table to the queueids state table.

The complications and difficulties that arose when parsing real-world log files are documented in §5.7 [p. 107]; some action descriptions refer to specific difficulties they address. The complications are documented in a separate section to avoid overwhelming the action descriptions.

If the log line has enough information to identify the correct connection or mail, each action will save all the data captured by the recognising rule's regex to the connection or mail; usually, log lines that lack identifying information will be processed by the `UNINTERESTING` action. Each action is passed the same arguments:

rule The recognising rule.

data The data captured from the log line by the rule's regex.

line The log line, separated into fields:

- **timestamp** The time the log line was written to the log file.
- **program** The name of the program that generated the log line.
- **pid** The Process Identifier (pid) of the process that generated the log line.
- **host** The hostname of the server the log line was generated on.
- **text** The remainder of the log line, i.e. the message logged by the program and recognised by the rule.

PLP's actions:

BOUNCE_CREATED Postfix 2.3 and subsequent versions log the creation of bounce messages, and this action processes those log lines. This action creates a new mail if necessary; if the mail already exists the unknown origin flag will be removed from it. To deal with complication §5.7.14 [p. 124], this action checks a cache of recent bounce mails, to avoid incorrectly creating bounce mails when log lines are out of order, and also marks the mail as a bounce notification.

CLEANUP_PROCESSING `postfix/cleanup` processes every mail that passes through Postfix; details of what it does are available in §D [p. 174]. This action saves all data captured by the rule's regex if the log line has not come after a timeout (see §5.7.8 [p. 118]); it also creates the mail if necessary, setting its unknown origin flag (see §5.7.9 [p. 119]).

CLONE Multiple mails may be accepted on a single connection, so each time a mail is accepted the connection's state table entry must be cloned and saved in the state tables under its queueid; if the original data structure was used then second and subsequent mails would overwrite one another's data.

COMMIT Enter the data from the mail into the database. Entry will be postponed if the mail is a child waiting to be tracked (§5.7.3 [p. 110]). Once entered into the database, the mail will be usually be deleted from the state tables, but deletion will be postponed if the mail is the parent of mail re-injected for forwarding (§5.7.3 [p. 110]).

CONNECT Process a remote client connecting: create a new connection, indexed by `postfix/smtpd` pid. If a connection already exists it is treated as a symptom of a bug in PLP, and the action will issue a warning containing the full contents of the existing connection plus the log line that has just been parsed.

DELETE Deals with mail deleted using Postfix's administrative command `postfix/postsuper`. This action adds a dummy recipient address if

required (see §5.7.13 [p. 123]), then invokes the `COMMIT` action to save the mail to the database.

DELIVERY_ERROR Process log lines indicating that an error occurred and the remote client disconnected; the log lines processed by this action will be followed by log lines processed by the `DISCONNECT` action, so all this action does is save data from the log line.

DELIVERY_REJECTED Postfix rejected a mail delivery attempt from the remote client. This is the action most frequently specified by rules, because so many different restrictions are used to reject delivery attempts. This action is quite simple: if the log line contains a queueid, save the data captured by the rule's regex to the mail identified by that queueid; otherwise save it to the connection identified by the pid in the log line.

DISCONNECT Invoked when the remote client disconnects, it enters the connection in the database if it has any useful data, performs any required cleanup, and deletes the connection from the state tables. This action deals with aborted delivery attempts (§5.7.5 [p. 113]).

EXPIRY If Postfix has not managed to deliver a mail after trying for five days, it will give up and return the mail to the sender. When this happens the mail will not have a combination of Postfix programs that passes the valid combinations check, implemented to deal with the complication described in §5.7.11 [p. 120]; this action tags the mail as having expired, so the `COMMIT` action will skip the valid combinations check.

MAIL_BOUNCED This action behaves in exactly the same way as the `SAVE_DATA` action; it saves all data captured by the recognising rule's regex, and does nothing more. It is a separate action to distinguish delivery attempts that bounce from other delivery attempts.

MAIL_DISCARDED Sometimes mail is discarded by Postfix, e.g. mail submitted locally on the server using `postfix/pickup` that is larger

than the limit configured by the administrator. This action is used for those mails; it invokes the `COMMIT` action, but is a separate action to simplify further analysis.

MAIL_QUEUED This action represents Postfix picking a mail from the queue to deliver. This action needs to deal with out of order log lines when mail is re-injected for forwarding; see §5.7.3 [p. 110] for details.

MAIL_SENT This action behaves in exactly the same way as the `SAVE_DATA` action; it saves all data captured by the recognising rule's regex, and does nothing more. It is a separate action to distinguish successful delivery attempts from other delivery attempts.

MAIL_TOO_LARGE When a remote client tries to send a mail over SMTP that is larger than the server accepts, the mail will be discarded by Postfix and the client informed of the problem. The mail may have been accepted and partially transferred; if so the parser will have a data structure that must be disposed of. See §5.7.7 [p. 116] for full details; although that describes timeouts, the processing is the same for mails that are too large.

PICKUP The `PICKUP` action corresponds to the `postfix/pickup` service processing a locally submitted mail. A new mail will usually be created, although out of order log lines may have caused it to already exist, as documented in §5.7.9 [p. 119].

POSTFIX_RELOAD When Postfix stops running or reloads its configuration, it kills all `postfix/smtpd` processes, requiring all of the connections in PLP's state tables to be cleaned up, entered in the database, and deleted from the state tables. Postfix probably kills all the other components too, but PLP is only affected by `postfix/smtpd` processes exiting.

SAVE_DATA Every action that can locate the correct data structure in the state tables saves any data captured by the recognising rule's regex to it. The `SAVE_DATA` saves data in this way but does not do anything else;

it is invoked for log lines that contain useful data but do not require any further processing.

SMTPD_DIED Sometimes a `postfix/smtpd` dies, is killed by a signal, or exits unsuccessfully; the associated connection must be cleaned up, entered in the database if it has enough data, and deleted from the state tables. Sometimes the connection will not have enough data to satisfy the database schema, so it cannot be entered into the database for future analysis; unfortunately this means that the small amount of data that has been gathered by PLP will be lost.

SMTPD_WATCHDOG `postfix/smtpd` processes have a watchdog timer to deal with unusual situations; after five hours the timer will expire and the `postfix/smtpd` will exit. This occurs very infrequently, because many other timeouts should occur in the intervening hours, e.g. timeouts for DNS requests or timeouts reading data from the client. The active connection for that `postfix/smtpd` must be cleaned up, entered in the database, and deleted from the state tables.

TIMEOUT The connection with the remote client timed out, so the mail being transferred must be discarded by Postfix. The mail may have been accepted: if so the parser will have a data structure to dispose of. See §5.7.7 [p. 116] for full details.

TRACK Track a mail when it is re-injected for forwarding to another mail server; this happens when a local address is aliased to a remote address (see §5.7.3 [p. 110] for a full explanation). `TRACK` will be called when dealing with the parent mail, and will create the child mail if necessary. `TRACK` checks if the child has already been tracked, either with this parent or with another parent, and issues appropriate warnings if so.

UNINTERESTING This action just returns successfully: it is used when a log line needs to be recognised to avoid warning about unrecognised log lines, but does not either provide any useful data to be saved or require any processing.

5.5.2 Adding New Actions

Adding new actions is not as simple as adding new rules, though care has been taken in the architecture and implementation to make adding new actions as painless as possible; one of the few limitations is that PLP is written in Perl, so new actions must also be written in Perl. The implementer writes a subroutine that accepts the standard arguments given to actions, and registers it with the framework by calling the framework's `add_actions()` subroutine. No other work is required from the implementer to integrate the action into the parser; all of their attention and effort can be focused on correctly implementing their new action. The action may need to extend the list of valid combinations described in §5.7.11 [p. 120] if the new action creates a different set of acceptable programs, but this would only be necessary if the new action processes log lines from Postfix components that PLP does not have rules for, e.g. `postfix/virtual` or `postfix/lmtp`. The new action must be registered before the rules are loaded, because it is an error for a rule to specify an unregistered action; this helps catch mistakes made when adding new rules.

5.6 Rules

The rules are responsible for recognising each log line and specifying the correct action to be invoked. The rules will be the most visible component in any parser based on this architecture, and also the component most likely to be modified by users. Rules need to be as simple as possible so that users can easily modify them or add new rules, but each parser must balance that simplicity with the need to provide enough flexibility and power to successfully parse inputs.

The role of the rules in the architecture is covered in detail in §4.4 [p. 61]; this section is concerned with the practical aspects of how rules are implemented and used in PLP. The structure of the rules table has already been documented in §5.3.2 [p. 79]; that description will not be repeated here, but should be fleshed out by the sample rule in §5.6.1 on the following page.

The process of creating new rules from unparsed log lines is dealt with in §5.6.2 [p. 100], followed by the algorithm used by the utility supplied with PLP that creates regexes from unparsed log lines; the regex components provided by PLP to ease writing of complex regexes are covered in §5.6.3 [p. 104]. The rule conditions used in PLP are the penultimate topic (§5.6.4 [p. 106]), and this section concludes with some suggestions for how to detect overlapping rules.

5.6.1 Sample Rule

The sample rule in table 5.1 on the following page recognises the log line that results from Postfix rejecting a delivery attempt because the domain part of the sender address does not have an A, AAAA, or MX DNS entry, i.e. mail could not be delivered to any address in the sender's domain (for full details see [17]). An example log line that would be recognised by this rule:

```
NOQUEUE: reject: RCPT from smtp.example.com[192.0.2.1]:
  550 5.1.8 <alice@example.com>:
  Sender address rejected: Domain not found;
  from=<alice@example.com> to=<bob@example.net>
  proto=SMTP helo=<smtp.example.com>
```

The attributes in table 5.1 on the next page are used as follows:

name, description, and restriction_name: are not used by the parser, they serve to document the rule for the user's benefit.

program and regex: program is used to restrict the log lines this rule will be used to recognise; see §5.6.4 [p. 106] for details. The regex does the actual recognition of log lines, and data captured by the regex (e.g. sender, recipient) will be automatically saved to the results and connections tables. The regex components used in the regex are described in §5.6.3 [p. 104].

Table 5.1: Sample rule used by PLP

Attribute	Value
name	Unknown sender domain
description	We do not accept mail from unknown domains
restriction_name	reject_unknown_sender_domain
program	`postfix/smtpd`
regex	`^__RESTRICTION_START__ <(__SENDER__)>: Sender address rejected: Domain not found; from=<\k<sender>> to=<(__RECIPIENT__)> proto=E?SMTP helo=<(__HELO__)>$`
result_data	
connection_data	
action	`DELIVERY_REJECTED`
hits	0
hits_total	0
priority	0
debug	0

action: The action to be invoked when the rule recognises a log line. See §5.5.1 [p. 92] for details of the actions implemented by PLP, and §4.3 [p. 59] for the role of actions in the architecture.

result_data and connection_data: are used to provide data not present in the log line, but are unused in this rule.

hits, hits_total, and priority: hits and priority are used when ordering the rules for more efficient parsing (see §6.1.2 [p. 137]). At the end of each parsing run hits is set to the number of log lines recognised by the rule. Hits_total is the sum of hits over every parsing run, but is otherwise unused by the parser.

debug enables or disables printing of debugging information when this rule recognises a log line.

5.6.2 Creating New Rules

The log files produced by Postfix differ from installation to installation, because administrators have the freedom to choose the subset of available restrictions that suits their needs, including using different DNSBL services, policy servers, or custom rejection messages. To ease the process of parsing new log lines, the architecture separates rules from actions: adding new actions requires some effort, but adding new rules to recognise new log lines is trivial, and occurs much more frequently.

To add a new rule a new row must be added to the rules table in the database, containing all the required attributes: action, name, description, program, and regex; all the other attributes are either optional, are set by the parser, or have sensible defaults. The name and description fields should be set based on the meaning of the log line, to help others understand which log lines this rule will recognise; the value for the program field will be obvious from the unrecognised log lines. The action depends on the what the log line represents, e.g. a delivery rejection, a mail being delivered, some useful information, or something else; examine the list of actions in §5.5.1 [p. 92] to determine the correct one. The regex needs to be based on the log line, but see below for a tool to ease the process of creating regexes from unrecognised log lines.

Other attributes may be required: connection_data, result_data, priority, or restriction_name. In general it will only become clear that connection_data or result_data are required when PLP warns about an entry in the connections or results tables that is missing some required fields, because values for those fields are not present in any of the log lines for that connection or mail. For example, the rule that recognises the `postfix/pickup` component processing a mail sets client_hostname to `localhost` and client_ip to `127.0.0.1`, because the mail originates on the local server. If the new rule deliberately uses the architecture's overlapping rules feature the priority field needs to be set, on this rule and possibly others; the priority field may be needed on unintentionally overlapping rules too, but that is more difficult to determine. Finally, the restriction_name field should be set if the rule's action is `DELIVERY_REJECTED`;

the name of the restriction should be clear from the content of the log line.

A program is provided with PLP to ease the process of creating new rules from unrecognised log lines, based on the algorithm developed by Risto Vaarandi for his Simple Logfile Clustering Tool (SLCT) [1]. The differences between the two algorithms will be outlined as part of the explanation below. The core of the SLCT algorithm is quite simple: programs generally create log lines by substituting variable words into a fixed pattern, and analysis of the frequency with which each word occurs can be used to determine whether the word is variable or part of the fixed pattern. This classification can be used to group similar log lines and generate a regex to match each group of log lines. The algorithm has five steps:

Pre-process the input. The modified algorithm begins by leveraging the knowledge gained from writing rules and developing PLP; it performs many substitutions on the input log lines, replacing commonly occurring variable terms (e.g. email addresses, IP addresses, the standard start of rejection messages) with keywords described in §5.6.3 [p. 104]. The purpose of this step is to utilise existing knowledge to create more accurate regexes; it replaces many variable words with fixed words, improving the subsequent classification of words as fixed or variable. Regex metacharacters in the log line will be escaped, to avoid generating invalid or incorrect regexes. The altered log lines are written to a temporary file, which the next stage will use instead of the original input file.

In the original algorithm the purpose of the pre-processing stage was to reduce the memory consumption of the program. In the first pass it generates a hash [14] (from a small range of values) for each word of each log line, incrementing a counter for each hash. The counters will be used in the next pass to filter out words.

Calculate word frequencies. The position of words within a log line is important: a word occurring in two log lines does not indicate similarity unless it occupies the same position in both log lines. If a variable term substituted into a log line contains spaces, it will appear to the algorithm

as more than one word. This will alter the position of subsequent words, so a word occurring in different positions in two log lines *may* indicate similarity, but the algorithm does not attempt to deal with this possibility. The modified algorithm maintains a counter for each `(word, word's position within the log line)` tuple, incrementing it each time that word occurs in that position.

The original algorithm hashes each word and checks that result's counter from the previous pass: if the word's hash does not have a high frequency, the word itself cannot have a high frequency, so it must be variable and does not need a counter maintained for the `(word, word's position within the log line)` tuple, reducing the number of counters required and thus the program's memory consumption. This reduces the number of counters maintained in this step, reducing memory requirements at the cost of increased CPU usage.

As time goes by, the amount of memory typically available to a program or algorithm increases, and the need to reduce memory requirements correspondingly decreases, so the modified algorithm omits the hashing step and maintains counters for all tuples. Most of the infrequently occurring words will have been substituted with keywords during the first step, vastly reducing the number of tuples to maintain counters for; the original algorithm does not have the detailed knowledge leveraged by the modified algorithm, because it is a generic tool.

Classify words based on their frequency. The frequency of each `(word, word's position within the log line)` tuple is checked: if its frequency is greater than the threshold supplied by the user (1% of all log lines is generally a good starting point), it is classified as a fixed word, otherwise it is classified as a variable term. If a variable term appears sufficiently often it will be misclassified as a fixed term, but that should be noticed by the user when reviewing the new regexes, and will be obvious when the new rules do not recognise some log lines they are expected to. Variable terms are replaced by `.+` to match one or more of any character.

Build regexes. The words are reassembled to produce a regex matching the log line, and a counter is maintained for each regex. Contiguous sequences of `.+` in the newly assembled regexes are collapsed to a single `.+`; any resulting duplicate regexes and their counters are combined. If a regex's counter is lower than the threshold supplied by the user the regex is discarded; this second threshold is independent of the threshold used to differentiate between fixed and variable words, but once again 1% of log lines is a good starting point.

Test the new regexes. The final step replaces keywords in the new regexes in the same way as PLP does, and compiles each regex to check they are valid. A match will be attempted between all of the new regexes and each of the original unparsed log lines (not the pre-processed log lines); the user will be warned if a log line is not matched by any regex, or if a log line is matched by more than one regex. The number of log lines each regex matches is counted, as is the number of log lines matched by all regexes, though a log line is counted once only, even if matched by more than one regex.

This step is not performed by the original algorithm.

The new regexes are displayed for the user to add to the ruleset, either as new rules or merged into the regexes of existing rules; also displayed are the number of unrecognised log lines each regex was expected to match, and the number it actually matched, to help the user notice problems in the new regexes. Discarding regexes will result in some of the unrecognised log lines not being matched; when the ruleset has been augmented with the new regexes, the original log files should be parsed again, and any remaining unparsed log lines used as input to this utility.

This utility is not expected to create perfect regexes, but it greatly reduces the effort required to deal with unrecognised log lines. The regexes it generates will be self-contained: a parser that relies on using cascaded parsing would require a modified algorithm, perhaps by replacing the pre-processing stage with one that applies existing cascading rules to each log line, and uses the resulting modified log lines for the remainder of the algorithm.

5.6.3 Regex Components

Each rule's regex will have keywords expanded when the ruleset is loaded, for several reasons:

- It simplifies both reading and writing of regexes and helps to make each regex largely self-documenting. For example, the meaning of __CLIENT_HOSTNAME__ is immediately clear, whereas its expansion `(?:unknown|(?:[-.\w]+))` needs to be deciphered each time it is encountered.

- It avoids needless repetition of complex regex components, and allows the components to be corrected or improved in one location. For example, __SENDER__ is used in 68 rules; if a mistake is discovered in it the mistake only needs to be corrected in one place.

- It enables automatic extraction and saving of captured data. The regex snippets use Perl 5.10's named capture buffers [15] to capture data, so the mapping between captures and fields does not need to be explicitly specified by the rule.

To improve efficiency, the keywords are expanded and every rule's regex is compiled before attempting to parse the log file, otherwise every regex would be recompiled each time it was used, resulting in a large, data dependent slowdown, as described in §6.1.5 [p. 145]. Most of the keywords are named after the fields in the connections or results tables they populate: __CLIENT_HOSTNAME__, __CLIENT_IP__, __DELAY__, __DELAYS__, __ENHANCED_STATUS_CODE__, __HELO__, __MESSAGE_ID__, __QUEUEID__, __RECIPIENT__, __SENDER__, __SERVER_HOSTNAME__, __SERVER_IP__, __SIZE__, and __SMTP_CODE__.

The other keywords need more explanation:

__CHILD__	The queueid of a child mail; see §5.7.3 [p. 110].
__COMMAND__	All SMTP commands.

__CONN_USE__	How many times the connection was reused; Postfix tries to reuse connections whenever possible to reduce the load on both the sending and receiving servers.
__DATA__	This snippet is special: it does not match anything by itself, so it must be followed by a pattern written by the rule author, but it captures whatever is matched by that pattern. For example, `/(__DATA__connection (?:refused|reset by peer))/` matches either "connection refused" or "connection reset by peer" and causes it to be saved to the data field of that log line's result.
__PID__	The pid of a `postfix/smtpd` process that dies or is killed; see the `SMTPD_DIED` action in §5.5.1 [p. 92].
__RESTRICTION_START__	Matches the standard information Postfix includes at the start of most log lines resulting from rejecting a delivery attempt.
__SHORT_CMD__	Postfix sometimes logs SMTP commands in a short, single word form; this snippet matches all of those, except `DATA`, which typically has a more specific rule. Priorities could have been used instead of excluding `DATA`.

Some similarity exists between regex components and cascaded parsing: each regex component resembles a rule that recognises part of a log line and consumes it, leaving the remainder to be recognised by other components or cascaded rules. The major difference between the two is that regex components are explicitly used by the author of the ruleset, whereas cascaded parsing would be dynamically applied by the framework.

5.6.4 Rule Conditions

Rule conditions as part of the architecture have already been documented in §4.4.2 [p. 63], and they can be very complex and difficult to evaluate. Complex conditions are not required in parsers based on this architecture: PLP uses quite simple rule conditions. Each rule has a `program` attribute that specifies the Postfix component whose log lines it recognises, and each log line contains the name of the Postfix component that produced it; when trying to recognise log lines, the framework will only use rules where the two are equal. This avoids needlessly trying rules that will not recognise the log line, or worse, might recognise it unintentionally. In addition the framework supports generic rules, whose program attribute is "*"; these will be used if none of the program-specific rules recognise the log line. If none of the rules are successful the framework will warn the user, informing them that they need to augment their ruleset, and alerting them that the results stored in the database may be incomplete because PLP failed to recognise some log lines.

5.6.5 Overlapping Rules

The advantages and difficulties of overlapping rules have already been addressed in §4.4.3 [p. 65] and will not be repeated here. PLP does not try to detect overlapping rules; that responsibility is left to the author of the rules. A mechanism is provided for ruleset authors to specify the order that overlapping rules will be tried in: the priority field in each rule. Negative priorities may be useful for catchall rules.

Detecting overlapping rules is difficult, but the following approaches may be helpful:

- Sort rules by program and regex e.g. with an SQL query similar to:

```
SELECT program, regex
    FROM rules
    ORDER BY program, regex;
```

  The rules are sorted first by program, then by regex, because rules

cannot overlap if their programs are different. Note that this query does not properly deal with generic rules whose program is *; those rules will be used on all log lines that have not been recognised by program-specific rules.

Inspect the list to see if any pair of regexes look suspiciously similar; overlapping regexes will often lie beside one another when sorted.

- Compare the results of parsing using different rule orderings, as described in §6.1.2 [p. 137]. Parse several log files using optimal ordering, then dump a textual representation of the rules, connections, and results tables. Repeat with shuffled and reversed ordering, starting with a fresh database. If the ruleset does not have overlapping rules the tables from each run will be identical; differences indicate overlapping rules. The rules that overlap can be determined by examining the differences in the tables: each result contains a reference to the rule that created it, which will change if that rule overlaps with another. Unfortunately this method cannot prove the absence of overlapping rules; it can detect overlapping rules, but only if the log files have log lines that are recognised by more than one rule.

5.7 Complications Encountered During Development of the Postfix Log Parser

It was initially expected that parsing Postfix log files would be a relatively simple task, requiring a couple of months of work. The author had found Postfix log files useful when investigating problems reported by users, and an examination of several log files gave the impression that they would be straightforward to parse, process, and understand. The large variation in log lines was not apparent, because most log lines are recognised by a small set of rules, as shown in figure 6.9 [p. 138]. Most of the myriad complications and difficulties documented in this section were discovered during PLP's development, but the first three complications were identified during the

planning and design phase of this project, influencing the architecture's design.

Each of these complications caused PLP to operate incorrectly, generate warning messages, or leave mails in the state table. The complications are listed in the order in which they were overcome during development of PLP, with the first complication occurring several orders of magnitude more frequently than the last. When deciding which problem to tackle next, the problem causing the highest number of warning messages or mails incorrectly remaining in the state tables was always chosen, because that approach yielded the biggest improvement in the parser, and made the remaining problems more apparent. Several of the complications were discovered because PLP is very careful to check the origin of each mail; if it does not know the origin of a mail, that mail will be tagged as fake, and a warning issued if the mail's origin has not been determined before trying to save it to the database. Some of the solutions to these complications require discarding log lines because they are out of order, and, less frequently, discarding a data structure containing data gathered about a connection or a mail, because of the paucity of information contained in the data structure.

The fifteen complications documented in this section can be divided into three broad classes:

Recreating Postfix behaviour Three complications needed to be solved for PLP to accurately recreate Postfix's behaviour.

Lack of information in the log files Seven complications are caused by lack of information in the log files. In general, some extra logging by Postfix could remove these complications, and for one complication (§5.7.4 [p. 112]), extra logging in later versions of Postfix does remove it; three of the complications would be removed if Postfix generated a log line whenever it discarded a mail.

The order that log lines are found in log files Five complications are caused by out of order log lines. Postfix is not to blame for these: they are caused by process scheduling, inter-process communication delays, and are more likely to occur when the mail server is heavily loaded.

5.7.1 Queueid Vs Pid

A delivery attempt lacks a queueid until the first recipient has been accepted, so log lines must first be correlated by `postfix/smtpd` pid, then transition to being correlated by their queueid. This is relatively minor, but does require:

- Two versions of several functions: `by_pid` and `by_queueid`.
- Two state tables to hold data structures.
- Each action needs to know whether it deals with accepted mails, and should find the mail in the `queueid` state table, or deals with delivery attempts, and thus will find the connection in the `connections` state table. Some actions, e.g. `DELIVERY_REJECTED`, need to query both tables. Support functions fall into two groups: one group deals with state tables, and so need to know which one to operate on; the functions in the other group are passed a specific mail or connection to work with, and do not need to know about state tables at all.

The `CLONE` action is responsible for copying an entry from the `connections` state table to the `queueids` state table.

5.7.2 Connection Reuse

Multiple independent mails may be delivered across one connection: this requires PLP to clone the connection's data as soon as a mail is accepted, so that subsequent mails will not overwrite each other's data. This must be done every time a mail is accepted, because it is impossible to tell in advance which connections will accept multiple mails. Once a mail has been accepted its log lines will not be correlated by pid any more: its queueid will be used instead, as described in §5.7.1. If the original connection has any useful data (e.g. rejections) it will be saved to the database when the client disconnects. One unsolved difficulty is distinguishing between different groups of rejections, e.g. when dealing with the following sequence:

1. The client attempts to deliver a mail, but it is rejected.

2. The client issues the RSET command to reset the SMTP session.
3. The client attempts to deliver another mail, likewise rejected.

Ideally, the sequence above would result in two separate entries in the connections table, but only one will be created.

5.7.3 Re-injected Mails

Mails sent to local addresses are not always delivered directly to a mailbox: sometimes they are sent to and accepted for a local address, but need to be delivered to one or more remote addresses because of aliasing. When this occurs, a child mail will be injected into the Postfix queue, but without the explicit logging that mails injected by `postfix/smtpd` or `postfix/postdrop` have. Thus the source of the mail is not immediately discernible from the log line in which the mail's queueid first appears: from a strictly chronological reading of the log lines it usually appears as if the child mail does not have an origin. Subsequently, the parent mail will log the creation of the child mail, e.g. parent mail `3FF7C4317` creates child mail `56F5B43FD`:

```
3FF7C4317:  to=<username@example.com>, relay=local,
    delay=0, status=sent (forwarded as 56F5B43FD)
```

The sample log line above would be processed by the `TRACK` action, which creates the child mail if it does not exist in the state tables, links it to the parent mail, and checks that the child is not being tracked for a second time.

Unfortunately, although all log lines from an individual process appear in chronological order, the order in which log lines from different processes are interleaved in a log file is subject to the vagaries of process scheduling. In addition, the first log line belonging to the child mail (the example log line above belongs to the parent mail) is logged by either `postfix/qmgr` or `postfix/cleanup`, so the order also depends on how soon they process the new mail.

Because of the uncertain order that log lines can appear in, PLP cannot complain when it encounters a log line from either `postfix/qmgr` or `postfix/cleanup` for a mail that does not exist in the state tables; instead it

must create the state table entry and flag the mail as coming from an unknown origin. Subsequently invoked actions will clear that flag if the origin of the mail becomes clear. The parser could omit checking where mails originate from, but requiring an explicit source helps to expose bugs in the parser; such checks helped to identify the complications described in §5.7.8 [p. 118] and §5.7.9 [p. 119].

Process scheduling can have a still more confusing effect: the child mail will often be created, delivered, and entirely finished with, *before* the parent mail logs its creation! Thus, mails flagged as coming from an unknown origin cannot be entered into the database when their final log line is processed, and PLP cannot warn the user; instead they must be marked as ready for entry and subsequently entered once their origin has been identified. Unfortunately, it is not possible to distinguish between child mails waiting to be tracked and other mails whose origin is unknown, except for bounce notifications, as described in §5.7.4 on the following page. Mails whose origin is unknown can remain in the state tables indefinitely if their origin is not determined at some point; they will cause a queueid clash if the queueid is reused, and, most importantly, PLP's understanding of such mails is incorrect. This problem has only been observed when a state table entry is missing its initial log line, usually because it is in an earlier log file; this specific instance is not a serious problem, because PLP cannot be expected to fully understand and reconstruct a mail when some of that mail's log lines are missing.

Tracking re-injected mail requires PLP to do the following in the `COMMIT` action:

1. If a mail is tagged with the unknown origin flag, it is assumed to be a child mail whose parent has not yet been identified. The mail is tagged as ready to be entered in the database, but entry is postponed until the parent is identified. The child mail will not have any subsequent log lines: only its parent will refer to it.

2. If the mail is a child mail whose parent has been identified, it is entered in the database as usual, then removed from its parent's list of children. If this child is the last mail on that list, and the parent has already

been entered in the database, the parent will be removed from the state tables.

3. If the mail is a parent, it is entered in the database as usual because there will be no further log lines for it. There may be child mails waiting to be entered in the database; these are entered as usual, and removed from the state tables. If the state tables contain incomplete child mails, the parent's removal from the state tables will be postponed until the last child has been entered.

5.7.4 Identifying Bounce Notifications

Postfix 2.2.x (and presumably previous versions) does not generate a log line when it generates a bounce notification; the log file will have log lines for a mail whose origin is unknown. Similarities exist to the problem of identifying re-injected mails discussed in §5.7.3 [p. 110], but unlike the solution described therein bounce notifications do not eventually have a log line that identifies their origin. Heuristics must be used to identify bounce notifications:

1. The sender address is `<>`.

2. Neither `postfix/smtpd` nor `postfix/pickup` have logged any messages associated with the mail, indicating that it was generated internally by Postfix, rather than accepted via SMTP or submitted locally by `postfix/postdrop`.

3. The message-id has a specific format:
   `YYYYMMDDhhmmss.queueid@server_hostname`
   e.g. `20070321125732.D168138A1@smtp.example.com`

4. The queueid embedded in the message-id must be the same as the queueid of the mail: this is what distinguishes a new bounce notification from a bounce notification that is being re-injected as a result of aliasing. For the latter, the message-id will be unchanged from the original bounce notification, and so even if it happens to be in the correct format, i.e. if

it was generated by Postfix on this or another server, it will not equal the queueid of the mail.

Once a mail has been identified as a bounce notification, the unknown origin flag is cleared and the mail can be entered in the database.

A small chance exists that a mail will be incorrectly identified as a bounce notification, because the heuristics used may be too broad. For this to occur the following conditions would have to be met:

1. The mail must have been generated internally by Postfix.
2. The sender address must be <>.
3. The message-id must have the correct format and contain the queueid of the mail. Although a mail sent from elsewhere could easily have the correct message-id format, the chance that the queueid in the message-id would correspond with the queueid of the mail is extremely small.

If a mail is misclassified as a bounce message it will almost certainly have been generated internally by Postfix; arguably, misclassification of this kind is a benefit rather than a drawback, because other mails generated internally by Postfix will be handled correctly. Postfix 2.3 and subsequent versions log the creation of a bounce message, so this complication does not arise in their log files. The solution to this complication will help with solving the complication in §5.7.14 [p. 124].

This check is performed during the `COMMIT` action.

5.7.5 Aborted Delivery Attempts

Some mail clients behave strangely during the SMTP dialogue: the client aborts the first delivery attempt after the first recipient is accepted, then makes a second delivery attempt for the same recipient that it continues with until delivery is complete. Microsoft Outlook is one client that behaves in this fashion; other clients may act in a similar way. An example dialogue exhibiting this behaviour is presented below (lines starting with a three digit number are sent by the server, the other lines are sent by the client):

```
220 smtp.example.com ESMTP
EHLO client.example.com
250-smtp.example.com
250-PIPELINING
250-SIZE 15240000
250-ENHANCEDSTATUSCODES
250-8BITMIME
250 DSN
MAIL FROM: <sender@example.com>
250 2.1.0 Ok
RCPT TO: <recipient@example.net>
250 2.1.5 Ok
RSET
250 2.0.0 Ok
RSET
250 2.0.0 Ok
MAIL FROM: <sender@example.com>
250 2.1.0 Ok
RCPT TO: <recipient@example.net>
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
The mail transfer is not shown.
250 2.0.0 Ok: queued as 880223FA69
QUIT
221 2.0.0 Bye
```

Postfix does not log a message making the client's behaviour clear, so heuristics are required to identify when a delivery attempt is aborted in this way. A list of all mails accepted during a connection is saved in the connection's state table entry, and the accepted mails are examined when the disconnection action is invoked. Each accepted mail is checked for the following:

- Was the second result processed by the `CLONE` action? The first two `postfix/smtpd` log lines will be a connection log line and a mail acceptance log line.
- Is `postfix/smtpd` the only Postfix component that produced a log line for this mail? Every mail that passes normally through Postfix will have a `postfix/cleanup` log line, and later a `postfix/qmgr` log line; lack of a `postfix/cleanup` log line is a sure sign the mail did not make it too far.
- Can the mail be found in the state tables? If not it cannot be an aborted delivery attempt.
- If third and subsequent results exist, were those log lines processed by the `SAVE_DATA` action? Any log lines after the first two should be informational only.

If all the checks above are successful the mail is assumed to be an aborted delivery attempt and is removed from the state tables. There will be no further log lines for such mails, so without identifying and discarding them they accumulate in the state table and will cause clashes if the queueid is reused. Such mails cannot be entered in the database because the only data they contain is the client hostname and IP address, but the database schema requires many more fields — see §5.3.3 [p. 82] and §5.3.4 [p. 83]. These heuristics are quite restrictive, and appear to have little scope for false positives; any false positives would cause a warning when the next log line for such a mail is parsed. False negatives are less likely to be detected: there may be queueid clashes (and warnings) if mails remain in the state tables after they should have been removed; if the queueid is not reused, the only way to detect false negatives is to examine the state tables after each parsing run.

This check is performed in the `DISCONNECT` action; it requires support in the `CLONE` action where the list of accepted mails is maintained.

5.7.6 Further Aborted Delivery Attempts

Some mail clients disconnect abruptly if a second or subsequent recipient is rejected; they may also disconnect after other errors, but such disconnections are handled elsewhere in the parser, e.g. §5.7.7. Postfix does not log a message saying the mail has been discarded, as should be expected by now. The checks to identify this happening are:

- Is the mail missing its `postfix/cleanup` log line? Every mail that passes normally through Postfix will have a `postfix/cleanup` log line; lack of one is a sure sign the mail did not make it too far.
- Were there three or more `postfix/smtpd` log lines for the mail? There should be a connection log line and a mail acceptance log line, followed by one or more delivery attempt rejected log lines.

If both checks are successful the mail is assumed to have been discarded by Postfix when the client disconnected; PLP will remove it from the state tables. There will be no further log lines for such mails, so if PLP does not deal with them immediately they accumulate in the state table and will cause clashes if the queueid is reused.

These checks are made during the `DISCONNECT` action.

5.7.7 Timeouts During DATA Phase

The DATA phase of the SMTP conversation is where the headers and body of the mail are transferred. Sometimes a timeout occurs or the connection is lost during the DATA phase;[1] when this occurs Postfix will discard the mail and PLP needs to discard the data associated with that mail. It seems more appropriate to save the mail's data to the database, but if a timeout occurs no data will be available to save; the timeout is recorded and saved with the connection instead. To deal properly with timeouts the `TIMEOUT` action does the following:

[1] For the sake of brevity *timeout* will be used throughout this section, but everything applies equally to lost connections.

1. Record the timeout and data extracted from the log line in the connection's results.

2. If no mails have been accepted over the connection, nothing needs to be done; the `TIMEOUT` action ends.

3. If one or more recipients have been accepted, Postfix will have allocated a queueid for the incoming mail, and there may be a mail in the state tables that needs to be discarded by PLP. The timeout may have interrupted transfer of an accepted delivery attempt, or it may have occurred after a mail delivery attempt was rejected. If a mail needs to be discarded, the following checks will all pass:

   - The timestamp of the log line preceding the timeout log line must be earlier than the timestamp of the last accepted delivery attempt, i.e. there have not been any rejections since then the delivery attempt was accepted.
   - The mail must exist in the state tables.
   - The mail must not have a `postfix/qmgr` log line.

   If all checks pass the mail will be discarded from the state tables and will not be entered in the database. If one or more checks do not pass, PLP assumes that the timeout happened after a rejected delivery attempt. This assumption is not necessarily correct, because Postfix could have accepted an earlier recipient, rejected a later one, and continued to accept delivery of the mail for the first recipient. In that case the timeout applies to the partially accepted mail, which will be discarded by Postfix and should be discarded by PLP; however, this has not occurred in practice. Processing timeouts is further complicated by the presence of out of order `postfix/cleanup` log lines: see §5.7.8 on the following page for details.

This complication is dealt with by the `TIMEOUT` action, with help from the `CLONE` action. When a client tries to delivery a mail larger than the sever

will accept, the `MAIL_TOO_LARGE` action will perform the same processing as the `TIMEOUT` action.

5.7.8 Discarding Cleanup Log Lines

The author has only observed this complication occurring after a timeout, though there may be other circumstances that trigger it. Sometimes the `postfix/cleanup` log line for a mail being accepted is logged after the timeout log line, by which time PLP has discarded the mail because it did not have enough data to satisfy the database schema (see §5.7.7 [p. 116]) and was not expected to have any more log lines; parsing the `postfix/cleanup` log line causes the `CLEANUP_PROCESSING` action to create a new state table entry, to help deal with re-injected mails (§5.7.3 [p. 110]). This is incorrect because the log line actually belongs to the mail that has just been discarded; if the queueid is reused there will be a queueid clash, otherwise the new mail will just remain in the state tables.

During the `TIMEOUT` action, if the mail's `postfix/cleanup` log line is still pending, the `TIMEOUT` action updates the `timeout_queueids` state table, adding the queueid and the timestamp from the log line. To deal with this complication, the following checks will be performed for each `postfix/cleanup` log line that is processed:

- Does the `timeout_queueids` state table have an entry for the queueid in the log line? If an entry is found it will be removed, regardless of whether the remaining criteria are satisfied. If the queueid does not exist in the `timeout_queueids` state table, the log line being processed cannot belong to a discarded mail.

- Has the queueid been reused yet, i.e. does it have an entry in the `queueids` state table? If it has an entry in the `queueids` state table, the log line being processed belongs to that mail, not to a previously discarded mail.

- The timestamp of the `postfix/cleanup` log line must be within ten minutes of the mail acceptance timestamp. Timeouts happen after five

minutes, but some data may have been transferred slowly, and empirical evidence shows that ten minutes is not unreasonable; it appears to be a good compromise between false positives (log lines incorrectly discarded) and false negatives (new state table entries incorrectly created).

The `postfix/cleanup` log line must pass the checks above for it to be discarded because some, but not all, connections where a timeout occurs will have an associated `postfix/cleanup` log line; if the `CLEANUP_PROCESSING` action blindly discarded the next `postfix/cleanup` log line after a timeout it would sometimes be mistaken.

This complication is handled by the `CLEANUP_PROCESSING` and `TIMEOUT` actions.

5.7.9 Pickup Logging After Cleanup

When mail is submitted locally, `postfix/pickup` processes the new mail and generates a log line. Occasionally, this log line will occur later in the log file than the `postfix/cleanup` log line, so the `PICKUP` action will find that a state table entry already exists for that queueid. Normally when this happens a warning is generated by the `PICKUP` action, but if the following conditions are met it is assumed that the log lines were out of order:

- The only program that has logged anything so far for the mail is `postfix/cleanup`.
- The difference between the timestamps of the `postfix/cleanup` and `postfix/pickup` log lines is less than five seconds.

As always with heuristics, there may be circumstances in which these heuristics match incorrectly, but none have been identified so far. This complication only seems to occur during periods of particularly heavy load, so is most likely caused by process scheduling vagaries.

This complication is dealt with by the `PICKUP` action.

5.7.10 Smtpd Stops Logging

Occasionally a `postfix/smtpd` will stop logging without an immediately obvious reason. After poring over log files for some time, several reasons have been found for this rare event:

1. Postfix is stopped or its configuration is reloaded. When this happens all `postfix/smtpd` processes exit, so all entries in the connections state table must be cleaned up, entered in the database if they have enough data, and deleted.

2. Sometimes a `postfix/smtpd` is killed by a signal (sent by Postfix, by the administrator, or by the OS), so its active connection must be cleaned up, entered in the database if it has enough data, and deleted from the connections state table.

3. Occasionally a `postfix/smtpd` will exit with an error, so the active connection must be cleaned up, entered in the database if it has enough data, and deleted from the connections state table.

4. Every Postfix process uses a watchdog timer that kills the process if it is not reset for a considerable period of time (five hours by default). This safeguard prevents Postfix processes from running indefinitely and consuming resources if a failure causes them to enter a stuck state.

The circumstances above account for all occasions where a `postfix/smtpd` suddenly stops logging. In addition to removing an active connection from the state tables, the last accepted mail may need to be discarded, as described in §5.7.7 [p. 116]; otherwise the queueid state table is untouched.

This complication is handled by several actions: `POSTFIX_RELOAD` (1), `SMTPD_DIED` (2 & 3), and `SMTPD_WATCHDOG` (4).

5.7.11 Out of Order Log Lines

Occasionally, a log file will have out of order log lines that cannot be dealt with by the various techniques described in §5.7.3 [p. 110], §5.7.8 [p. 118],

or §5.7.9 [p. 119]. In the 93 log files used in §6 [p. 129] this problem occurs only five times in 60,721,709 log lines. All five occurrences have the same characteristics: the `postfix/local` log line showing delivery to a user's mailbox comes after the `postfix/qmgr` log line showing removal of the mail from the queue because delivery is complete. This causes several problems: the data in the state tables for the mail is not complete, so it cannot be entered into the database; a new mail is created when the `postfix/local` log line is processed and remains in the state tables; four warnings are issued per pair of out of order log lines.

The `COMMIT` action examines the list of programs that have produced log lines for each mail, checking the list against a table of known-good Postfix component combinations. If the mail's combination is found in the table it can be entered in the database; if the combination is not found entry must be postponed and the mail flagged for later entry. The `MAIL_DELIVERED` action checks for that flag; if the log line is has just processed has caused the mail to reach a valid combination then entry of the mail into the database will proceed, otherwise it must be postponed once more.

The valid combinations are explained below. In addition to the components shown in each combination, every mail will have log lines from both `postfix/cleanup` and `postfix/qmgr`, and any mail may also have a log line from `postfix/bounce`, `postfix/postsuper`, or both.

`postfix/local`: Local delivery of a bounce notification, or local delivery of a re-injected mail.

`postfix/local`, `postfix/pickup`: Mail submitted locally on the server, delivered locally on the server.

`postfix/local`, `postfix/pickup`, `postfix/smtp`: Mail submitted locally on the server, for both local and remote delivery.

`postfix/local`, `postfix/smtp`, `postfix/smtpd`: Mail accepted from a remote client, for both local and remote delivery.

`postfix/local`, `postfix/smtpd`: Mail accepted from a remote client, for local delivery only.

`postfix/pickup`, `postfix/smtp`: Mail submitted locally on the server, for remote delivery only.

`postfix/smtp`: Remote delivery of either a re-injected mail or a bounce notification.

`postfix/smtp`, `postfix/smtpd`: Mail accepted from a remote client, then remotely delivered. Typically this is a mail server relaying mail for clients on the local network to addresses outside the local network.

`postfix/smtpd`, `postfix/postsuper`: Mail accepted from a remote client, then deleted by the administrator before any delivery attempt was made. Note that `postfix/postsuper` is required, not optional, for this combination.

This check applies to accepted mails only, not to rejected mails. This check is performed during the `COMMIT` action, with support from the `MAIL_DELIVERED` action.

5.7.12 Yet More Aborted Delivery Attempts

The aborted delivery attempts described in §5.7.5 [p. 113] occur frequently, but those described in this section only occur four times in the 93 log files used in §6 [p. 129]. The symptoms are the same as in §5.7.5 [p. 113], except that the mail *has* a `postfix/cleanup` log line; nothing can be found in the log files to explain why this mail does not have any further log lines. The only way to detect these mails is to periodically scan all mails in the state tables, deleting any mails with the following characteristics:

- The timestamp of the last log line for the mail must be 12 hours or more earlier than the timestamp of the last log line parsed from the current log file.
- There must be exactly two `postfix/smtpd` and one `postfix/cleanup` log lines for the mail, with no additional log lines.

12 hours is a somewhat arbitrary time period, but it is far longer than Postfix would delay delivery of a mail in the queue, unless it was not running for an extended period of time. Each time the end of a log file is reached, the state tables are scanned for mails matching the characteristics above, and any mails found are deleted.

5.7.13 Mail Deleted Before Delivery is Attempted

Postfix logs the recipient address when delivery of a mail is attempted, so if delivery has yet to be attempted PLP cannot determine the recipient address or addresses. This can occur when mail is accepted faster than Postfix can attempt delivery, and the administrator deletes some of the mail before Postfix has had a chance to attempt delivery. The deleted mail's recipient address or addresses will not have been logged yet, and the deleted mails will not have any more log lines. A dummy recipient address needs to be added by PLP, because every mail is required by the database schema (§5.3.4 [p. 83]) to have at least one recipient. When this complication occurs the log file will typically show many instances of it, closely grouped. Generally, this problem arises because the administrator has deleted some mails from Postfix's mail queue to stop a mail loop.

This lack of information cannot easily be overcome: it is simple to configure Postfix to log a warning for every accepted recipient, but Postfix will not yet have allocated a queueid for the mail when the warning for the first recipient is logged, so the warning will be associated with the connection rather than the accepted mail. A queueid will be allocated after Postfix accepts the MAIL FROM command if `smtpd_delay_open_until_valid_rcpt` is set to "no", but that setting will cause disk IO for every delivery attempt, instead of just for delivery attempts where recipients are accepted, and consequently a drastic reduction in the performance of the mail server.

The `DELETE` action is responsible for handling this complication.

5.7.14 Bounce Notification Mails Delivered Before their Creation Is Logged

This is yet another complication that only occurs during periods of extremely high load, when process scheduling and even hard disk access times cause strange behaviour. In this complication, bounce notification mails are created, delivered, and deleted from the queue, *before* the log line from `postfix/bounce` that explains their origin. The origin of the mail is unknown when it is created, but when the time comes to enter it into the database its origin is correctly identified by the heuristics used to identify Postfix 2.2 bounce notifications, described in §5.7.4 [p. 112]. To avoid incorrectly creating a new mail when the out of order bounce notification log line is processed, the `COMMIT` action maintains a cache of recently committed bounce notification mails named `bounce_queueids`, which the `BOUNCE_CREATED` action subsequently checks when processing the bounce creation log line. If the queueid exists in the cache, and its start time is less than ten seconds before the timestamp of the bounce log line, it is assumed that the bounce notification mail has already been processed and the `BOUNCE_CREATED` action does not create one. If the queueid exists in the cache it is removed, because it has either just been used or the problem did not occur for the new mail. Whether the `BOUNCE_CREATED` action creates a new mail or finds an existing mail in the `queueids` state table (not the `bounce_queueids` cache), it flags the mail as having been seen by the `BOUNCE_CREATED` action; if this flag is present the `COMMIT` action will not add the mail to the `bounce_queueids` cache. This prevents a bounce notification log line being incorrectly discarded if a queueid is reused within 10 seconds.

5.7.15 Mails Deleted from the Mail Queue During Delivery

The administrator can delete mails using `postfix/postsuper`; occasionally, mails that are in the process of being delivered will be deleted by the administrator. This results in the log lines from the delivery agent (`postfix/local`,

`postfix/smtp`, or `postfix/virtual`) appearing in the log file *after* the mail has been removed from the state tables and saved in the database. The `DELETE` action adds deleted mails to a cache named *postsuper_deleted_queueids*, which is checked by the `MAIL_DELIVERED` action, and the current log line discarded if the following conditions are met:

1. The queueid is not found in the state tables.
2. The queueid is found in the cache of deleted mails.
3. The timestamp of the log line is within 5 minutes of the final timestamp of the mail.

Sadly, this solution involves discarding some data, but the complication only arises eight times in quick succession in one log file, which is not in the 93 log files used for evaluating the parser; if this complication occurred more frequently it might be desirable to find the mail in the database and add the log line to it.

5.7.16 Summary

This section has described the complications encountered while implementing PLP. Five of the fifteen complications are caused by log lines appearing in an unexpected or abnormal order in log files, often because of heavy system load: these complications typically caused warnings and a new state table entry. Three of the fifteen complications must be dealt with to correctly recreate Postfix's behaviour. The remaining seven complications are caused by deficiencies in Postfix's logging, and some could easily be resolved by adding additional logging to Postfix; these seven are dealt with by applying heuristics to specific log lines or mails, sometimes in conjunction with information cached when mails are removed from the state tables.

5.8 Limitations and Possible Improvements

Every piece of software suffers from some limitations, and almost always has room for improvement. Below are the limitations and possible improvements that have been identified in PLP.

1. Each new Postfix release requires writing new rules or modifying existing rules to cope with the new or changed log lines. Similarly, using a new DNSBL, a new policy server, or new administrator-defined rejection messages also requires new rules.

2. The hostname used in the HELO command is not logged if the incoming delivery attempt is successful. Configuring Postfix to log the HELO hostname for accepted mails is relatively simple; create a restriction that is guaranteed to warn for every accepted mail, as follows:

   (a) Create `/etc/postfix/log_helo.pcre` containing:
   ```
   /^/    WARN Logging HELO
   ```

   (b) Modify `smtpd_data_restrictions` in `/etc/postfix/main.cf` to contain:
   ```
   check_helo_access pcre:/etc/postfix/log_helo.pcre
   ```

   Although `smtpd_helo_restrictions` seems like the natural place to log the HELO hostname, when it is evaluated for the first recipient there will not yet be a queueid allocated for the delivery attempt, so the log line will be associated with the connections rather than the mail. A queueid is guaranteed to have been allocated when the DATA command has been reached, and thus the queueid will be logged by any restrictions taking effect in `smtpd_data_restrictions`, and the log line can be associated with the correct mail. Specifying a HELO-based restriction in `smtpd_data_restrictions` does not cause any problems; Postfix will perform the check correctly.

   Logging the HELO hostname in this fashion also partially prevents the complication described in §5.7.13 [p. 123] from occurring, but only

when the mail has a single recipient. When a mail has a single recipient address it will be logged, but when a mail has multiple recipients no addresses are logged.

3. PLP will not detect that it is parsing the same log file twice, resulting in the database containing duplicate entries.

4. PLP does not distinguish between log files produced by different servers when parsing; all results will be saved to the same database. This may be viewed as an advantage, because log files from different servers can be combined in the same database, or it may be viewed as a limitation because it is impossible to distinguish between log files from different servers in the same database. If the results of parsing log files from different servers must remain separate, PLP can easily be instructed to use a different database.

5. Further complications may arise when parsing log files, and PLP will need to be modified to deal with them.

6. PLP does not limit the size of the database, which will grow without bounds unless the user deletes connections and results from it. This is both a benefit and a limitation: the benefit is that data will never be unexpectedly deleted, but the limitation is that the user must manage the size of the database.

5.9 Summary

This chapter has presented PLP, the parser implemented for this project, beginning with the assumptions made during its development. A simplified flowchart shows the most common paths taken through Postfix and PLP, accompanied by a description of the stages and transitions. A database provides storage for rules and for data gathered from log files; any further use of that data is dependent on a clear understanding of the database schema, so the role of the database schema as an API is described, followed by a diagram and a detailed description of the database schema. The framework

is documented next, including the steps it takes during initialisation, the parsing process, and the conveniences it offers to users. The performance data collected by the framework is described, as are the ways in which various optimisations can be disabled to demonstrate their effect, and the debugging options the framework provides. The implementation of actions in PLP is documented, including how frequently each action is specified by rules, and brief descriptions of why some actions are more popular than others. All the actions that are part of PLP are described in detail, followed by the process of adding a new action to the parser. The final component of the architecture, the rules, is also the most visible component, and its implementation is examined in detail with: a sample rule and an explanation of how every field in that rule is used; a description of how to add new rules and determine the values that should be used for each field; an explanation of the algorithm used by the utility that creates new regexes from unparsed log lines, and how it differs from the original algorithm it is based on. How PLP uses rule conditions and overlapping rules is discussed, accompanied by a description of the regex snippets the framework provides to simplify the regexes used in rules. The many complications and difficulties encountered while writing PLP, and the solutions developed to overcome them, are documented in depth; also documented are how solutions interact, and which action or actions each solution is implemented in. This chapter concludes with a list of the limitations identified in PLP. The next chapter will evaluate this implementation, examining both PLP's efficiency and the coverage it achieves when parsing log files.

Chapter 6

Evaluation

This chapter evaluates Postfix Log Parser (PLP) on two criteria: efficiency, and coverage of Postfix log files. Efficiency is important because PLP is intended to be used in a production environment, and must be capable of parsing log files generated by a high volume mail server in a reasonable period of time. Full coverage of Postfix log files is important because the data gathered by the parser must be accurate and complete for it to be useful and reliable. Progress in achieving one goal usually comes at the expense of the other: every extra measure implemented to improve accuracy slows down parsing. Some of the complications described in §5.7 [p. 107] occur fewer than ten times when parsing the 93 log files used to evaluate PLP in this chapter, yet their solutions slow down parsing of every log file. However, those solutions are retained in PLP, because accurate but slow parsing is preferable to sloppy but quick parsing.

The performance evaluation begins by describing the characteristics of the mail server that produced the 93 log files used to evaluate the parser's performance and coverage; the computer that the tests were run on is also described. The characteristics of the 93 log files are described next, with an explanation of why PLP has better performance when parsing the larger log files in the group. The effect of using different rule orderings is explored, and optimal rule ordering is compared to an oracle that allows the parser to use only one rule when recognising each log line. When PLP is used by other mail

administrators, they will need to extend the ruleset to parse their own log lines, so the next section addresses the question of how parser performance is affected by an increase in the number of rules in the ruleset. The penultimate topic is the simple optimisation of caching the results of compiling each rule's Regular Expression (regex), and the huge effect it has on parser efficiency. The performance evaluation concludes by examining where parsing time is spent: recognition of log lines or their subsequent processing by actions?

The second criterion the parser is evaluated on is its coverage of Postfix log files. Two kinds of coverage need to be evaluated: what proportion of log lines are correctly recognised by the ruleset, and what proportion of mail delivery attempts are correctly understood and reconstructed by the actions? The former is a requirement for the latter to be achieved, and the latter is important because the data provided by PLP must be both complete and correct for it to be of use to others.

6.1 Parser Efficiency

Efficiency is an obvious concern when the parser routinely needs to parse large log files. The mail server that generated the 93 log files used in this chapter is a production mail server handling mail for a university department; the benefit of using this mail server is that its log files exhibit the idiosyncrasies and peculiarities that a mail server in the wild must deal with, but the downside is that significantly altering its Postfix configuration to accommodate this project is not an option. Graph 6.1 on the next page shows the number of mails accepted over Simple Mail Transfer Protocol (SMTP), with mean and standard deviation in table 6.1 on the following page; as expected, far more mails are accepted on weekdays than at weekends. Note that mail generated on the server (e.g. bounce notifications, mail re-injected for forwarding, mail sent from mailing lists) does not contribute to these figures: in particular, the vast amounts of mail resulting from the mail loops noticeable in later graphs are not reflected in these figures.

The standard deviation for mails received on weekend days is quite high, because approximately four times the usual number of mails were received on

the weekend contained in log files 16 & 17. The standard deviation for all days is high because of the unusually high number of mails received during log files 16, 17, 35, 36, 39 & 40. Interestingly, the extra mails received during log files 35, 36, 39 & 40 stopped during log files 37 & 38, which may indicate that the source was a virus infected office machine that was turned off over the weekend. Unfortunately, the source cannot be investigated properly because all of the mails in question were relayed via the department's secondary Mail Transfer Agent (MTA) (a backup mail server, hosted elsewhere), so the log files showing the original source were unavailable. Relaying spam mail through a secondary MTA is a common practice amongst spam senders, because historically, primary MTAs had better anti-spam defences than secondary MTAs.

Graph 6.1: Number of mails received via SMTP per day

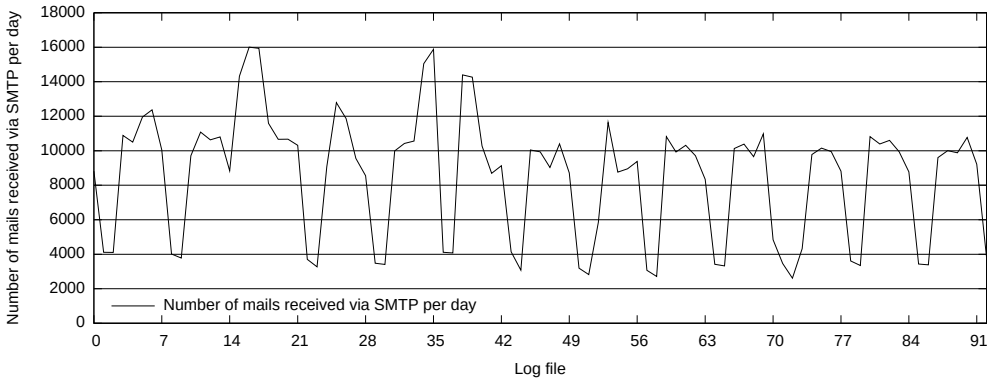


Table 6.1: Number of mails received via SMTP per day

	Mean	Std. Dev.
All days	8537.742	3562.945
Weekend days	4365.259	3091.683
Week days	10244.667	1985.398

When generating the timing data used in this section, 93 log files were used, each containing one day's log lines; the three months of contiguous log files contain 60.722 million log lines and total 9.385 GB. Each log file was parsed 10 times, and the mean parsing time calculated for each log file. The computer used for test runs was a Dell Optiplex 745, with components shown

Table 6.2: Details of the computer used to generate statistics

Component	Component in use
CPU	One dual core 2.40GHz Intel® Core™2 CPU, with 32KB L1 cache and 4MB L2 cache.
RAM	2GB 667 MHz DDR RAM.
Hard disk	One Seagate Barracuda 7200 RPM 250GB SATA disk.

in table 6.2; it was dedicated to the task of gathering statistics from test runs, and did not run any other programs while test runs were in progress. Saving results to the database was disabled for the test runs, because that dominates the run time of the parser, and the tests are aimed at measuring the speed of PLP rather than the speed of the database and the disks it is stored on. Parsing all 93 log files in one run without saving results to the database took 2 hours, 44 minutes, and 18.704 seconds, with mean throughput of 58.487 MB (369,551.887 log lines) parsed per minute. In contrast, when saving results to the database, parsing all 93 log files took 14 hours, 20 minutes, and 48.205 seconds, with mean throughput of 11.164 MB (70,540.738 log lines) parsed per minute — parsing takes up only 19.088% of execution time when saving results to the database.

6.1.1 Characteristics of the 93 Log Files

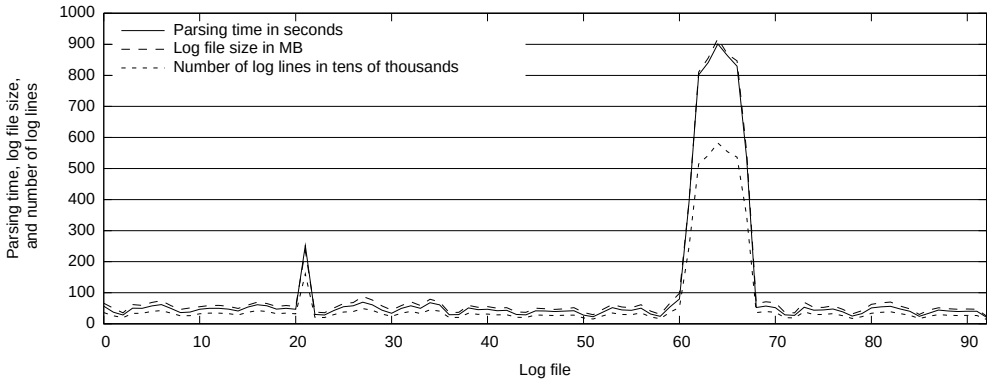
The 93 log files used in this chapter were generated on the School of Computer Science and Statistics' mail server from 2007/01/26 to 2007/04/28. The 93 log files have fairly consistent sizes and contents, except for two groups of log files: 22 & 62–68. Median log file size is 53.297 MB, containing 321,357 log lines, with a median of 10,088 mails accepted each weekday; large scale mail servers would accept many more mails and consequently generate much larger log files. Graph 6.2 [p. 134] shows the parsing time in seconds, log file size in MB, and number of log lines in tens of thousands, for each of the 93 log files. Parsing time is plotted against number of log lines in graph 6.4 [p. 134], and against log file size in graph 6.3 [p. 134]; the points on both graphs appear to

form straight lines, but they actually curve downwards slightly as they move to the right.

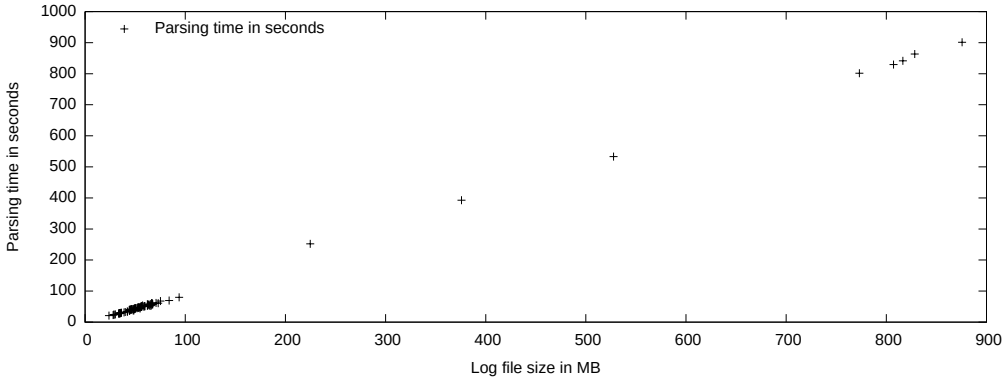
Graph 6.5 [p. 135] plots both the ratio of log file size to parsing time, and the ratio of number of log lines to parsing time (higher is more efficient in both cases); table 6.3 [p. 135] shows mean and standard deviation. The ratios are quite tightly banded, except they increase (i.e. improve) for log files 22 & 62–68, despite their larger than usual size. The log files in both groups are much larger than usual because of mail loops caused by users who set up mail forwarding incorrectly, resulting in a very different distribution of log lines: normally most log lines are generated by mail delivery attempts from other hosts, but during the mail loops most of the log lines resulted from failed delivery of bounce notifications re-injected for forwarding. The Postfix components that generated most of the log lines during the mail loops have fewer associated rules than the Postfix components whose log lines normally make up the bulk of each log file, so the mean number of rules used per log line is lower for these log files, as is the mean parsing time per log line. Table 6.4 [p. 135] shows the number of rules for each Postfix component; graph 6.6 [p. 136] shows the drop in the mean number of rules used per log line for log files containing a mail loop, and graph 6.7 [p. 136] shows the mean number of rules used per log line for each Postfix component for 93 log files.

The three plots on graph 6.2 on the following page are quite close together, and the graph can be difficult to read. Log files 62–68 show a large gap between log file size and number of log lines, whereas those plots appear to have a small gap for all the other log files, suggesting that the ratio of number of log lines to log file size is lower for log files 62–68 (i.e. fewer log lines per MB). This appearance is misleading: graph 6.8 [p. 136] shows the mean log line size for all 93 log files, and it reduces for log files 22 & 62–68, i.e. there are more log lines per MB in those log files.

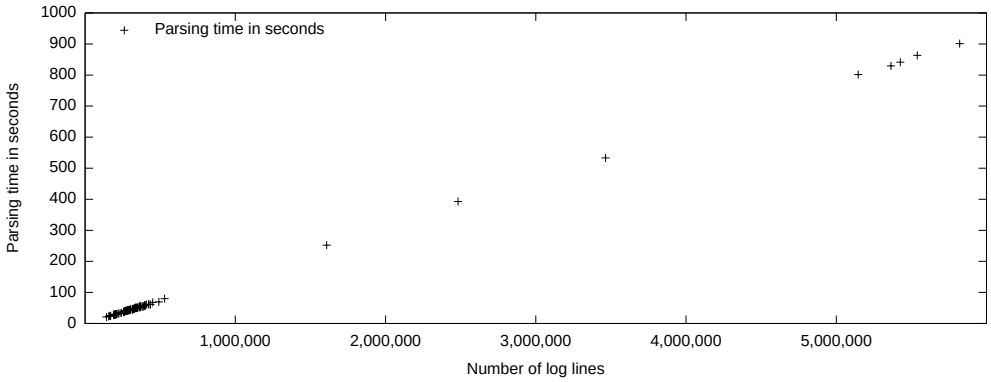
Graph 6.2: Parsing time, log file size, and number of log lines for 93 log files



Graph 6.3: Parsing time plotted against log file size



Graph 6.4: Parsing time plotted against number of log lines



Graph 6.5: Ratio of log file size and number of log lines to parsing time (higher is more efficient)

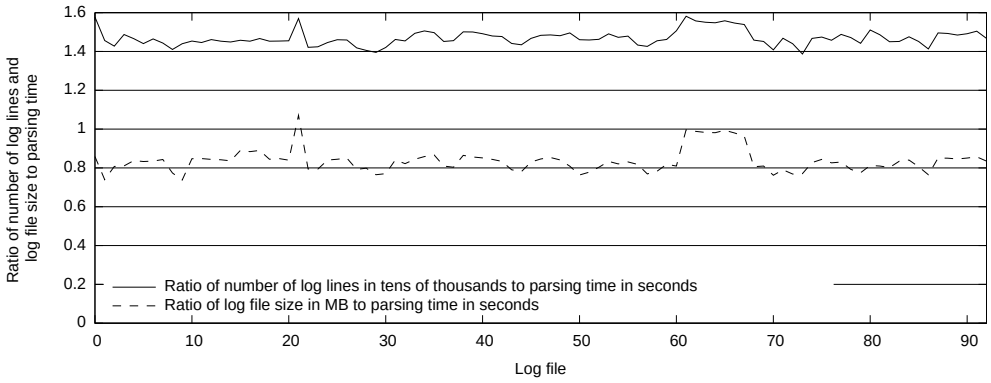


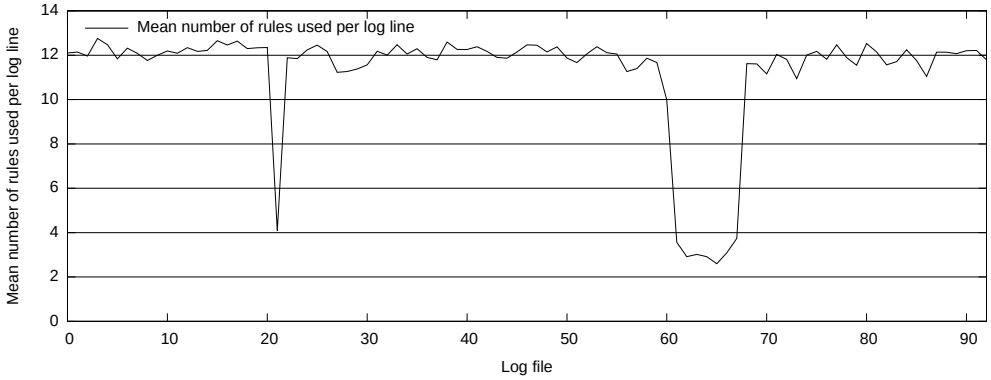
Table 6.3: Ratio of log file size and number of log lines to parsing time

	Mean	Std. Dev.
Log file size vs. parsing time (all log files)	0.836	0.059
Log file size vs. parsing time (log files 22 & 62–68)	0.995	0.030
Log file size vs. parsing time (other log files)	0.821	0.033
No. of log lines vs. parsing time (all log files)	1.470	0.039
No. of log lines vs. parsing time (log files 22 & 62–68)	1.556	0.013
No. of log lines vs. parsing time (other log files)	1.461	0.030

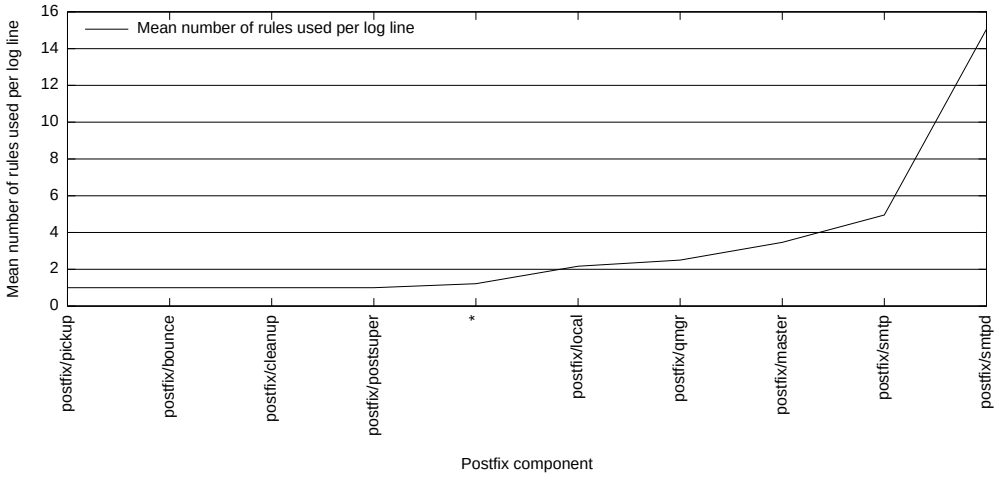
Table 6.4: Number of rules per Postfix component

Postfix component	Number of rules
*	4
postfix/bounce	1
postfix/cleanup	10
postfix/local	20
postfix/master	6
postfix/pickup	1
postfix/postsuper	6
postfix/qmgr	12
postfix/smtp	42
postfix/smtpd	82

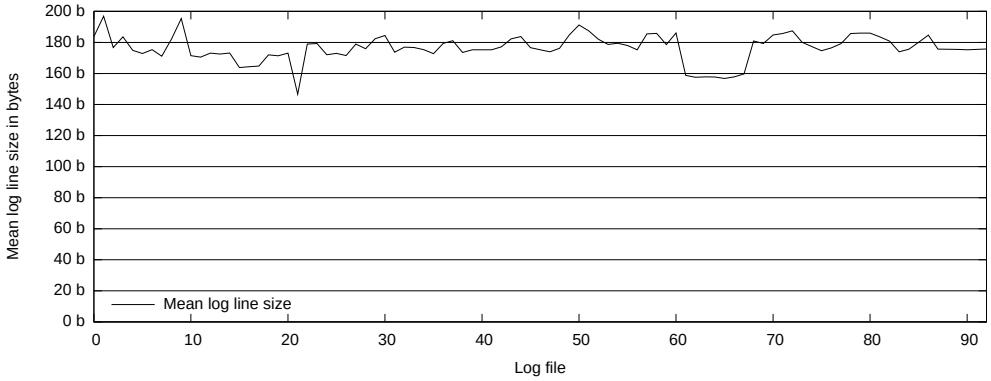
Graph 6.6: Mean number of rules used per log line



Graph 6.7: Mean number of rules used per log line for each Postfix component over 93 log files



Graph 6.8: Mean log line size in bytes



6.1.2 Rule Ordering for Efficiency

PLP's ruleset has 184 rules: the top 10 rules recognise 85.036% of the log lines in the 93 log files, with the remaining log lines split across the other 174 of the rules, as shown in graph 6.9 on the following page. Assuming that the distribution of log lines is reasonably consistent across log files, PLP's efficiency should benefit from using rules that recognise log lines more frequently before it uses rules that recognise log lines less frequently. To test this hypothesis, three full test runs were performed with different rule orderings:

Optimal The optimal order, according to the hypothesis: rules that recognise log lines most frequently will be used first.

Shuffled This ordering is intended to represent a randomly ordered ruleset. The rules will be shuffled once before use and will retain that ordering until the parser exits. Note that the ordering will change every time the parser is executed, so 10 different rule orderings will be generated for each log file in the test run.

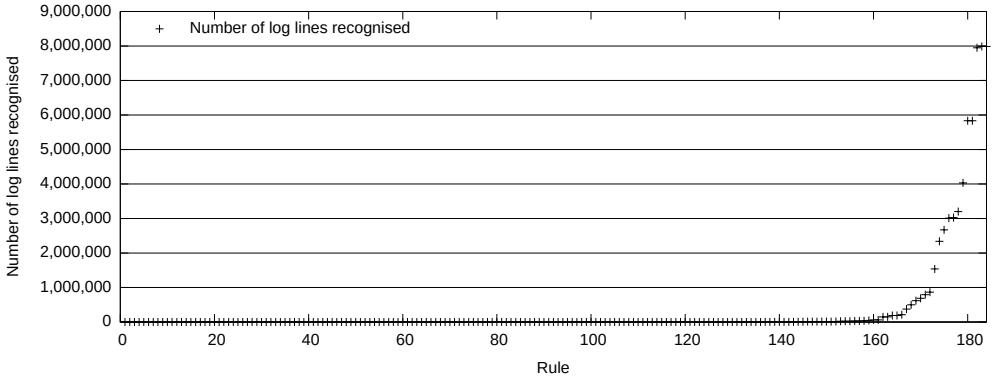
Reverse Hypothetically the worst order: rules that recognise log lines most frequently will be used last.

The parsing times of the three orderings are plotted against log file size in graph 6.10 on the next page; graph 6.11 [p. 139] shows the parsing times of optimal and reverse orderings relative to shuffled ordering, with mean and standard deviation in table 6.5 [p. 139]. Graph 6.12 [p. 139] shows the number of rules used by each ordering while parsing the 93 log files. This optimisation provides a mean reduction in parsing time of 30.743% with normal log files, making it a very worthwhile and effective optimisation.

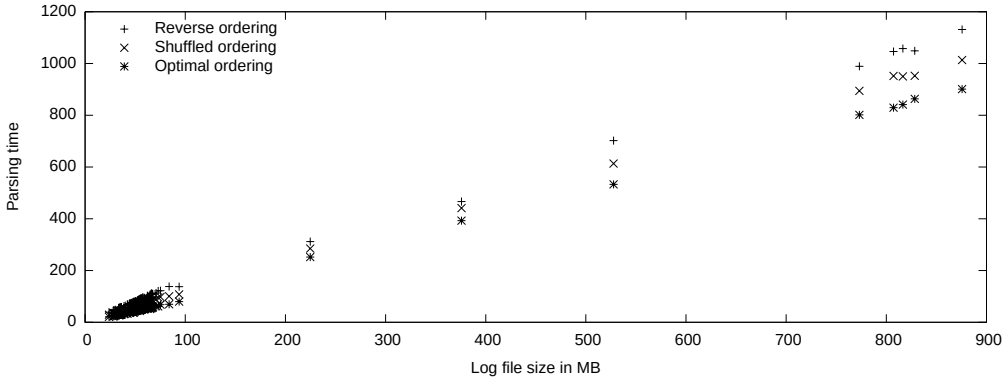
Table 6.5 [p. 139] shows that differences in rule ordering have less effect on parsing time when parsing log files 22 & 62–68, because of the different distribution of log lines in those log files. A careful examination of graph 6.11 [p. 139] shows that, for the first log file only, optimal and reverse orderings perform identically: this is because the hits field of each rule is zero for the

first log file, so optimal and reverse orderings produce identical rule orderings. For the first log file, shuffled ordering is the most efficient of the three, but that is accidental and cannot be relied upon.

Graph 6.9: Log lines recognised per rule



Graph 6.10: Parsing time plotted against log file size for optimal, shuffled, and reverse orderings



Graph 6.11: Parsing time of optimal and reverse orderings relative to shuffled ordering

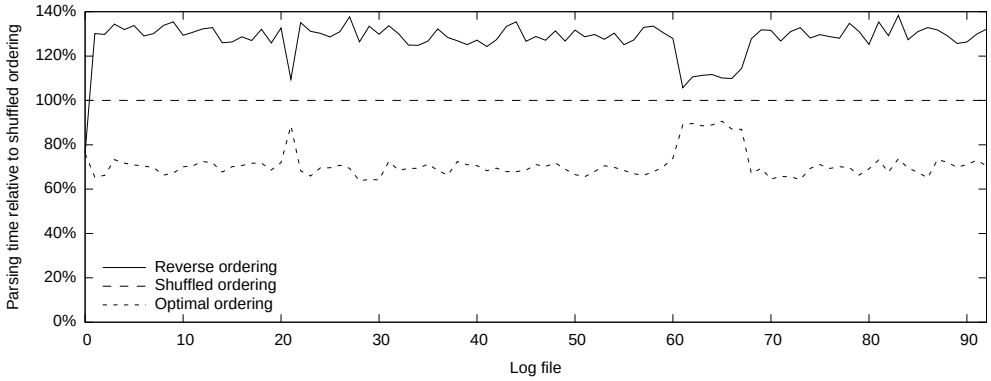
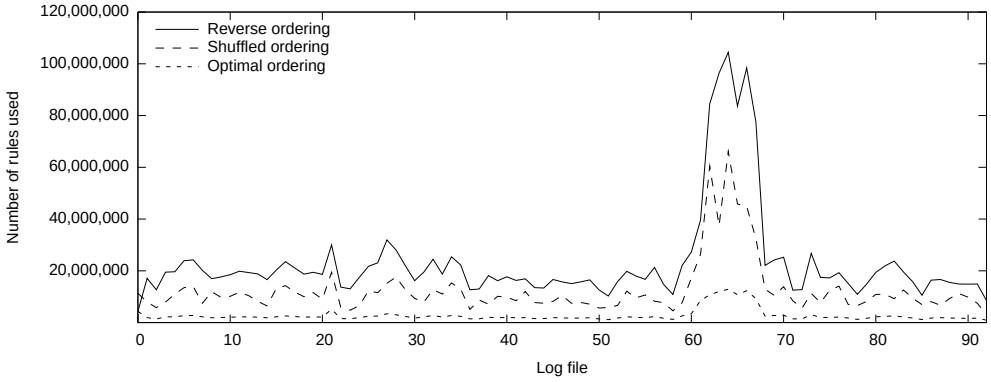


Table 6.5: Parsing time of optimal and reverse orderings relative to shuffled ordering

	Mean	Std. Dev.
Optimal ordering (all log files)	70.926%	5.977%
Optimal ordering (log files 22 & 62–68)	88.665%	1.133%
Optimal ordering (other log files)	69.257%	2.562%
Reverse ordering (all log files)	127.726%	8.266%
Reverse ordering (log files 22 & 62–68)	110.412%	2.289%
Reverse ordering (other log files)	129.356%	6.587%

Graph 6.12: Number of rules used by optimal, shuffled, and reverse orderings



6.1.3 Comparing Optimal Ordering Against an Oracle

Optimal rule ordering, described in §6.1.2 [p. 137], is the best rule ordering it is possible to achieve without having an oracle that magically divines which rule should be used to recognise each log line. Such an oracle would give perfect performance, because only one rule would need to be used to recognise each log line. PLP can save a list showing which rule recognised each log line, and use that list to simulate an oracle and improve parsing speed the *second* time a log file is parsed. This does not provide a practical benefit, but it does provide a means to evaluate the performance of optimal rule ordering in comparison to an oracle.

Graph 6.13 on the following page shows how the oracle and optimal ordering perform relative to shuffled ordering, with mean and standard deviation in table 6.6 on the next page; graph 6.14 on the following page shows parsing time plotted against log file size for the oracle, optimal ordering, and shuffled ordering. As expected, the oracle is more efficient than optimal ordering, but not by much, particularly when parsing the larger log files. Graph 6.15 [p. 142] shows the percentage increase in parsing time when using optimal ordering instead of the oracle, with mean and standard deviation in table 6.7 [p. 142].

Once again, the difference between the oracle and optimal ordering is at its lowest when parsing log files resulting from a mail loop (log files 22 & 62–68), because the mean number of rules used per log line is lower when parsing these log files (see graph 6.6 [p. 136]). The performance of optimal ordering relative to the oracle is much worse when parsing the first log file than for the remainder of the log files, because the hits field of every rule starts at zero, so optimal ordering does not provide any benefit for the first log file; the oracle, in contrast, is flawless for every log file.

Optimal ordering proves to be quite efficient: table 6.7 [p. 142] shows that optimal ordering is less than 9% slower than parsing using an oracle that magically divines the correct rule to use for each log line.

Graph 6.13: Parsing time of the oracle and optimal ordering relative to shuffled ordering

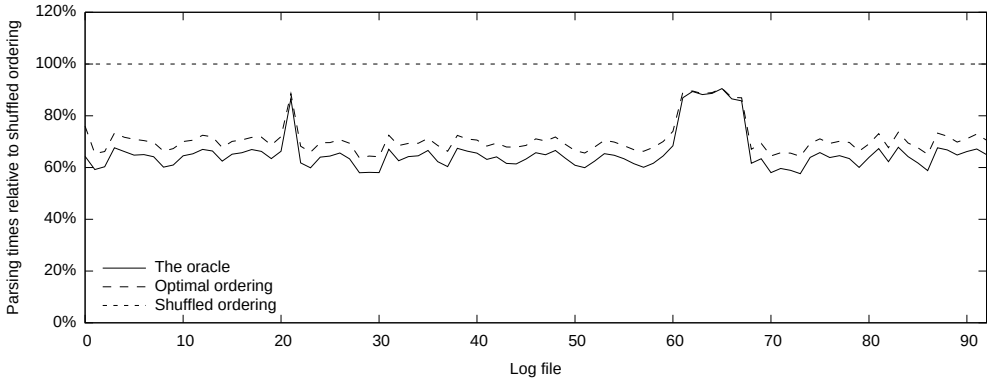
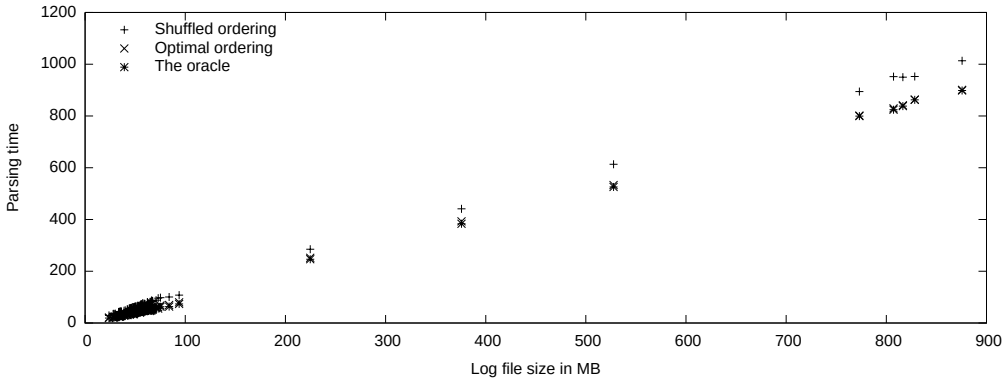


Table 6.6: Parsing time of the oracle and optimal ordering relative to shuffled ordering

	Mean	Std. Dev.
The oracle (all log files)	65.671%	7.300%
The oracle (log files 22 & 62–68)	87.829%	1.528%
The oracle (other log files)	63.586%	2.745%
Optimal ordering (all log files)	70.926%	5.977%
Optimal ordering (log files 22 & 62–68)	88.665%	1.133%
Optimal ordering (other log files)	69.257%	2.562%

Graph 6.14: Parsing time plotted against log file size for the oracle, optimal ordering, and shuffled ordering



Graph 6.15: Percentage increase in parsing time when using optimal ordering instead of the oracle

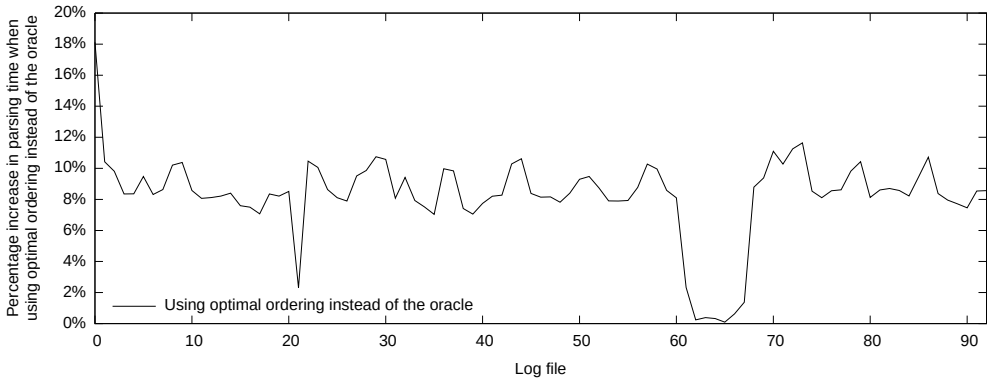


Table 6.7: Percentage increase in parsing time when using optimal ordering instead of the oracle

	Mean	Std. Dev.
Percentage increase (all log files)	8.269%	2.655%
Percentage increase (log files 22 & 62–68)	0.962%	0.865%
Percentage increase (other log files)	8.956%	1.464%

6.1.4 Scalability as the Ruleset Grows

How any architecture scales as the number of rules increases is important, but it is particularly important for this architecture because it is anticipated that the typical parser will have a large ruleset. The full PLP ruleset has 184 rules, whereas the minimum ruleset required to parse the 93 log files has 115 rules, 62.500% of the full ruleset. The full ruleset is larger because PLP is tested with 774 log files (2 years, $1\frac{1}{2}$ months of log files); testing with more log files increases the chance of finding bugs in the parser or new complications to be overcome. The 93 log files were each parsed 10 times using the minimum ruleset, and the mean parsing times compared to those generated using the full ruleset: the percentage parsing time increase when using the full ruleset instead of the minimal ruleset for optimal, shuffled, and reverse orderings is shown in graph 6.16 on the next page, with mean and standard deviation in table 6.8 [p. 144]. For each ordering, the parsing time using the maximum

ruleset is compared to the parsing time using the minimum ruleset: the three orderings are not compared to a common baseline.

Clearly, the increased number of rules causes a noticeable performance decrease with reverse ordering, and a lesser decrease with shuffled ordering, whereas optimal ordering shows scant change. Log files 22 & 62–68 show much smaller increases in parsing time than other log files do, because most of the log lines in those log files are produced by Postfix components with few rules, so removing unnecessary rules has little effect on the total number of rules used; table 6.9 on the next page shows the number of rules per Postfix component for each ruleset. Once again, graph 6.16 shows that optimal and reverse orderings have identical performance for the first log file, because the hits field of every rule starts at zero, and so the two rule orderings are identical for the first log file.

The optimal ordering shows a mean increase of just 1.368% in parsing time for a 60.000% increase in the number of rules. These results show that both the architecture and PLP scale extremely well as the ruleset increases in size, and that optimally ordering the rules makes an important contribution to this scalability.

Graph 6.16: Percentage parsing time increase when using the maximum ruleset instead of the minimum ruleset

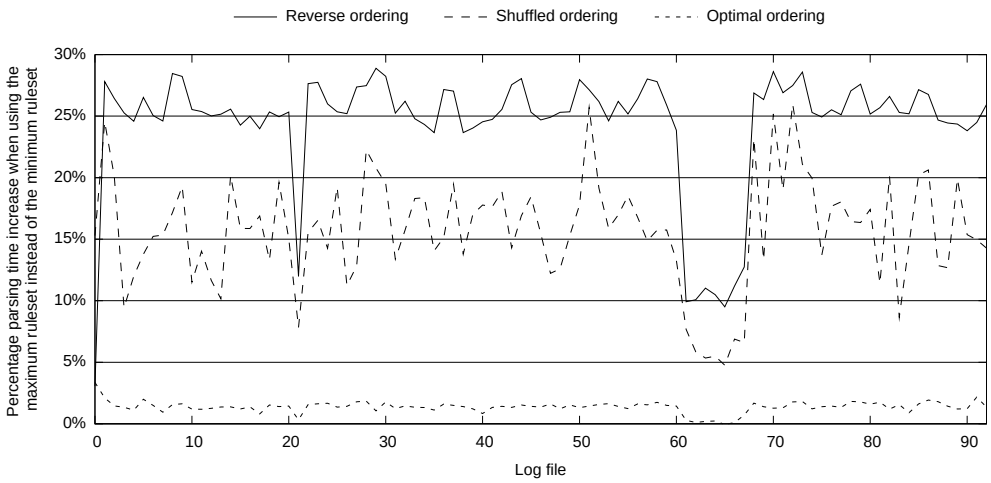


Table 6.8: Percentage parsing time increase when using the maximum ruleset instead of the minimum ruleset

	Mean	Std. Dev.
Optimal ordering (all log files)	1.368%	0.474%
Optimal ordering (log files 22 & 62–68)	0.237%	0.211%
Optimal ordering (other log files)	1.475%	0.331%
Shuffled ordering (all log files)	15.666%	4.461%
Shuffled ordering (log files 22 & 62–68)	6.304%	1.047%
Shuffled ordering (other log files)	16.547%	3.555%
Reverse ordering (all log files)	24.382%	4.949%
Reverse ordering (log files 22 & 62–68)	10.869%	1.025%
Reverse ordering (other log files)	25.654%	2.811%

Table 6.9: Number of rules per Postfix component in the maximum and minimum rulesets

Postfix component	Maximum ruleset	Minimum ruleset	Difference
*	4	3	1
postfix/bounce	1	1	0
postfix/cleanup	10	3	7
postfix/local	20	12	8
postfix/master	6	4	2
postfix/pickup	1	1	0
postfix/postsuper	6	3	3
postfix/qmgr	12	6	6
postfix/smtp	42	31	11
postfix/smtpd	82	51	31

6.1.5 Caching Compiled Regexes

Before the Perl interpreter attempts to match a regex against a piece of text, the regex is compiled into an internal representation and optimised to improve the speed of matching. This compilation and optimisation takes CPU time: for most regexes it takes far more CPU time than the actual matching does. If the interpreter is certain that a regex will not change while the program is running, it will automatically cache the results of compiling and optimising the regex for later use. The results of compiling a dynamically generated regex can be cached and used in preference to the original regex, but it is the responsibility of the programmer to do so; PLP does this with every rule's regex when the rules are loaded from the database.

A test run using optimal ordering was performed as described in §6.1 [p. 130], with one difference: regexes were not compiled and cached when the ruleset was loaded from the database, so the Perl interpreter had to compile each regex each time it was used when trying to recognise a log line. Graph 6.17 on the following page shows the effect that not caching compiled regexes has on parser performance, with mean and standard deviation in table 6.10 on the next page. For typical log files, the mean increase in parsing time when not caching compiled regexes is 558.945%; looking at it from the opposite direction, caching compiled regexes reduces parsing time by 84.824%. Caching compiled regexes is probably the single most effective optimisation possible in PLP, and was quite simple to implement: the framework compiles each rule's regex when the ruleset is loaded, and uses the compiled regex instead of the source regex when recognising log lines. As seen previously, log files 22 & 62–68 do not suffer such a large increase in parsing time when the optimisation is disabled; this is because, on average, fewer rules are used per log line, so fewer regexes are compiled per log line for those log files.

The increase in parsing time when parsing the first log file is much greater than for the other log files (see graph 6.17 on the following page); again, this is because every rule's hits field starts at zero, so optimal ordering is less efficient than usual, the mean number of rules used for each log line is higher than usual, and more regexes will need to be compiled when recognising

each log line. Parsing time is plotted against log file size for caching and not caching compiled regexes in graph 6.18; the plot for not caching compiled regexes is quite uneven compared to caching compiled regexes — some log files are particularly slow, whereas others are not as badly affected, but these differences have not been investigated.

Graph 6.17: Percentage increase in parsing time when not caching compiled regexes

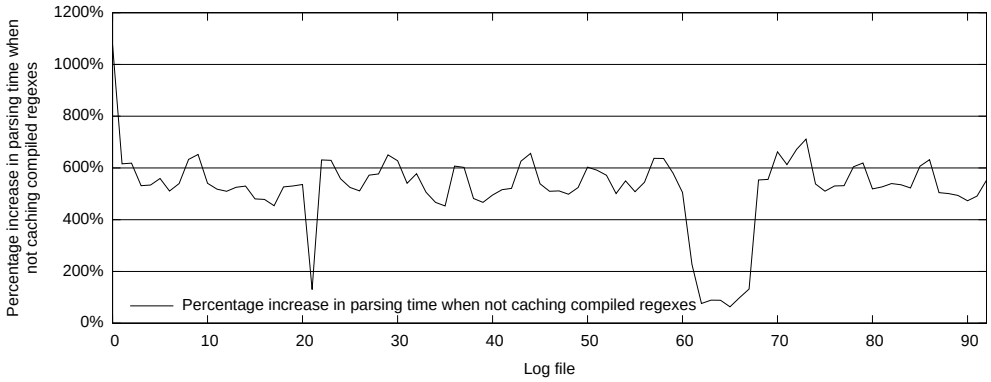
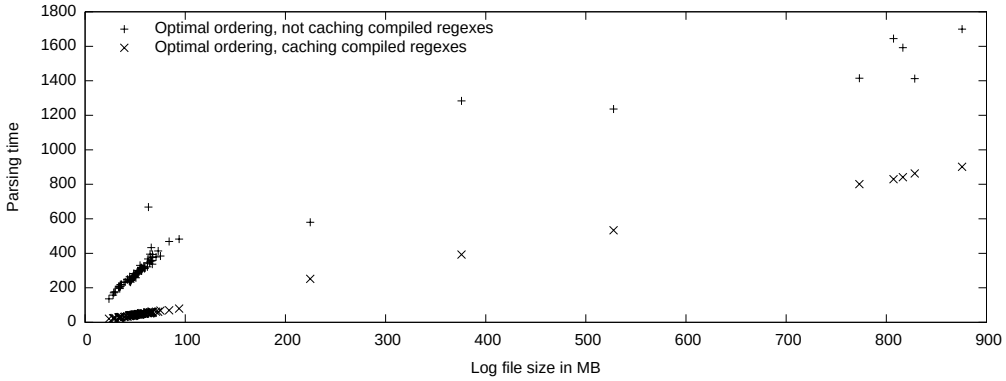


Table 6.10: Percentage increase in parsing time when not caching compiled regexes

	Mean	Std. Dev.
Not caching compiled regexes (all log files)	520.592%	147.229%
Not caching compiled regexes (log files 22 & 62–68)	113.096%	48.396%
Not caching compiled regexes (other log files)	558.945%	79.978%

Graph 6.18: Parsing time plotted against log file size when caching and not caching compiled regexes



6.1.6 Where is Parsing Time Spent: Recognising or Processing Log Lines?

The optimisations described in this chapter have addressed the process of recognising log lines, but have not optimised actions at all. Optimisation efforts have concentrated on recognition of log lines for two reasons:

1. Even with the optimisations described in this chapter enabled, recognising log lines still accounts for two thirds of the parser's execution time. Graph 6.19 on the following page shows the percentage of parsing time spent recognising log lines for each of the 93 log files, with mean and standard deviation shown in table 6.11 on the next page; for normal log files, 67.794% of parsing time is spent recognising log lines. To measure how long recognition of log lines takes, a full test run using optimal ordering and caching compiled regexes was performed as previously described in §6.1 [p. 130], but with one difference: actions were not invoked when a log line was recognised, so the parsing time did not include the time normally taken by invocation of actions; the mean parsing times for each log file were subtracted from the mean parsing times from a normal test run to obtain the parsing time taken by actions. The parsing times exclude the time taken for framework initialisation, loading and saving state tables, loading the ruleset, and other housekeeping tasks: in as far as possible, the times are just for recognising log lines or recognising log lines plus invoking actions.

   Optimal ordering reduces parsing time by 30.743% relative to shuffled ordering; actions occupy 32.206% of parsing time, less if any optimisations are disabled, so optimising actions could not possibly provide as big a performance increase as optimally ordering rules. Similarly, caching compiled regexes provides an 84.824% reduction in parsing time; not invoking actions at all reduces the most optimised parsing time by less than half that. Optimising actions to 1% of their original parsing time would be only slightly more effective than optimal ordering is, but would be vastly more difficult to implement; optimising actions

would not reduce parsing time enough to justify the amount of effort required.

2. The process of recognising log lines is not parser-specific (excluding evaluation of rule conditions), so the optimisations described in this chapter are applicable to all parsers based on this architecture. Actions are parser-specific, so it is unlikely that any optimisations made to actions would be portable to other parsers.

Individual actions and the framework could and have been optimised, but plenty of existing literature is available on the topic of optimising programs, so the subject is not dealt with here.

Graph 6.19: Percentage of parsing time spent recognising log lines

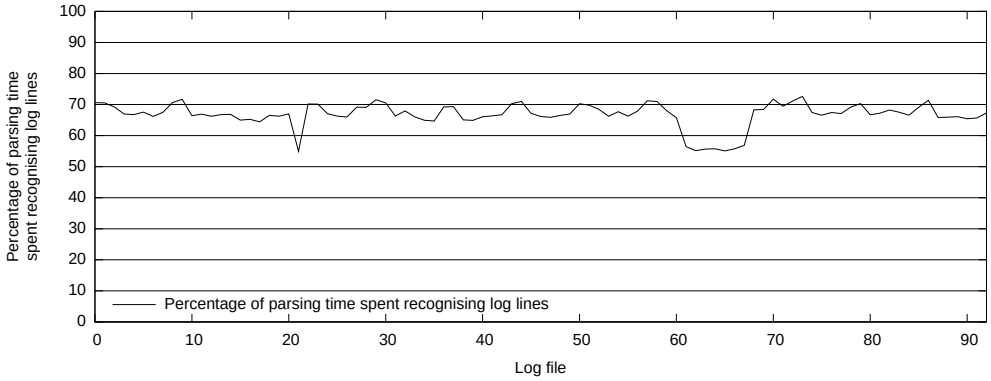


Table 6.11: Percentage of parsing time spent recognising log lines

	Mean	Std. Dev.
Recognising log lines (all log files)	66.753%	3.902%
Recognising log lines (log files 22 & 62–68)	55.701%	0.634%
Recognising log lines (other log files)	67.794%	2.010%

6.2 Coverage

The discussion of PLP's coverage of Postfix log files is separated into two parts: log lines correctly recognised, and mail delivery attempts correctly understood — the former is a requirement for the latter to be achieved. Correctly understanding and reconstructing every mail delivery attempt is important so that the information in the database is accurate and complete. Improving the proportion of log lines correctly recognised is the less difficult of the two, because it just requires new rules to be written or existing rules to be extended. Improving the proportion of correctly understood and reconstructed mail delivery attempts is more difficult and intrusive, because it requires adding or changing actions, and it can be much harder to realise that a deficiency exists and needs to be addressed.

6.2.1 Log Lines Correctly Recognised

Parsing a log line is a three step process:

1. Skip the log line if the ruleset does not contain any rules for the Postfix component that produced it.
2. Try each rule that recognises log lines from that Postfix component, then any generic rules, until a recognising rule is found; if the log line is not recognised, issue a warning and move on to the next log line.
3. Invoke the action specified by the recognising rule.

Each Postfix component whose log lines are of interest must have at least one rule that recognises its log lines, or all of its log lines will be silently skipped; in the extreme case of an empty ruleset the parser would skip every log line. PLP skips log lines from programs that do not have any associated rules, because there may be any number of log lines from other programs in the log file, and some Postfix components do not produce any log lines of interest. PLP does not parse log lines from non-Postfix programs, e.g. Amavisd-new or SpamAssassin; it could easily be extended to do so, if a method could be developed to correctly associate such log lines with existing state table entries.

To correctly recognise all log lines that are not skipped, there must be a rule to recognise each log line variant produced by each Postfix component; if a log line is not recognised the parser will issue a warning, to inform the user that they need to extend their ruleset. Each rule's regex should be as specific and precise as possible, to ensure accurate parsing: a rule with a regex that matches zero or more of any character will recognise every log line, but not in a meaningful way; most log lines contain fixed strings, so this is not a problem in practice.

Full coverage of log lines can be achieved without undue effort, yet once achieved it requires maintenance. Maintaining full coverage is an ongoing task because the set of log line variants changes over time, e.g. administrators add restrictions with custom messages, DNS Blacklist (DNSBL) messages change, or major releases of Postfix change log lines (usually by adding more information). Warnings are issued for any log lines that are not recognised; no warnings are issued for unrecognised log lines while parsing the 93 log files, so it can be safely concluded that zero false negatives arise. False positives are harder to quantify, short of examining each of the 60,721,709 log lines and checking that the correct rule recognised it. However, a random sample of 6039 log lines was parsed, and the results manually verified by inspection to ensure that the correct rule recognised each log line. The sample was generated by running the command:

```
perl -n -e 'print if (rand 1 < 0.0001)' LOG_FILES
```

to randomly extract roughly one log line in every 10,000 (it actually extracted 0.00994% instead of 0.010%). Each log line was examined and the correct rule identified from the 184 rules in the database; the correct rule was then compared to the rule that recognised the log line when parsing. The sample results contained zero false positives, and this check has been automated to ensure continued accuracy. Based on these results, and how precise each rule's regex is, the author is confident that zero false positives occur when parsing the 93 log files.

6.2.2 Mail Delivery Attempts Correctly Understood and Reconstructed

The proportion of mail delivery attempts that are correctly understood and reconstructed is more difficult to determine accurately than the proportion of log lines that are correctly recognised. The parser can dump its state tables in a human readable form; examining those tables with reference to the log files and the database is the best way to detect mails that were not reconstructed properly; many of the complications documented in §5.7 [p. 107] were detected in this way. PLP issues warnings when it detects any errors or discrepancies, e.g. when a queueid is reused but the previous mail remains in the state tables, when a queueid or Process Identifier (pid) is not found in the state tables, or when an entry in the state tables does not include sufficient data to satisfy the database schema. The parser should produce few or no warnings during parsing, and when finished parsing the state tables should only contain entries for mails that have log lines in subsequent log files. There will often be warnings about a missing queueid or pid when parsing the first few thousand log lines, because some earlier log lines for those connections or mails are in a previous log file; loading state tables saved when parsing the log file containing those log lines will solve this problem.

5 warnings are produced when parsing the 93 log files to generate the data used in this chapter, but because PLP errs on the side of producing more warnings rather than fewer, those 5 warnings represent 3 instances of 1 problem: 3 connections that started before the first log file, so their initial log lines are missing, leading to warnings when their remaining log lines are parsed. None of the warnings are false positives.

The state tables will contain entries for mails not yet delivered when the parser finishes parsing the log file. Ideally, those are the only entries the state tables will contain, though they may also contain mails whose initial log lines are not contained in the log files. Any other entries in the state tables are evidence of either a failure in parsing, or an aberration in the log files. After parsing the 93 log files, the state tables contain 18 entries:

- 1 connection that started only seconds before the log files ended and

the mail had not yet been fully transferred from client to server.

- 1 mail that had been accepted only seconds before the log files ended and had not yet been delivered.
- 9 mails whose initial log lines were not present in the log files. Six of those mails did not produce warnings because they resemble child mails waiting to be tracked with a parent; see §5.7.3 [p. 110] for details. The other three mails were missing more log lines, and so they produced five warnings, as documented previously.
- 7 mails that had yet to be delivered because of repeated delivery failures.

All of the mails remaining in the state tables have valid reasons for being present, so it can be concluded that zero false negatives occur when parsing the 93 log files. Once again, determining the false positive rate is much harder, because manually checking the results of parsing 13,850,793 connections and mails accepted, rejected, bounced, or delivered is infeasible. Considerable evidence exists that the false positive rate is extremely low, if not zero:

- PLP performs many checks to detect known problems, e.g. a queueid missing from the state tables. No such warnings are produced during the test runs other than the five described previously.
- Queueids and pids naturally identify log lines belonging to one mail or connection respectively; it is extremely unlikely that a log line would not be associated with the right connection.
- When dealing with the complications described in §5.7 [p. 107], the solutions are as specific and restrictive as possible, with the goal of minimising the number of false positives. In addition, the solution to the complication described in §5.7.11 [p. 120] imposes conditions that every reconstructed mail must comply with to be acceptable, not just the five mails exhibiting that complication.
- Every effort has been made to make PLP as precise, demanding, and particular as possible.

Figure 6.20: The command used to extract the log segment used to verify PLP's parsing

```
perl -Mstrict -Mwarnings -e '
    while (<>) {
        if (rand 1 < 0.0001) {
            my $count = 0;
            while ($count < 6000) {
                print scalar <>;
                $count++;
            }
            exit;
        }
    }' LOG_FILES
```

Verifying by inspection that the parser correctly processes all 13,850,793 mail delivery attempts in the 93 log files is infeasible, but verifying the parsing of a sample from those log files is a tractable albeit extremely time consuming task. A sample of log lines was obtained by randomly selecting a block of 6000 contiguous log lines from the 93 log files (0.00988% of the total number of log lines), using the command shown in figure 6.20. It is important that the log lines are contiguous, so that all log lines are present for as many of the mail delivery attempts contained in the block as possible. This log segment was parsed with all debugging options enabled, resulting in 167,448 lines of output.[1] All 167,448 lines were examined in conjunction with the log file segment and a dump of the resulting database, verifying that PLP recognised each of the log lines with the correct rule and invoked the correct action, which in turn correctly processed the log line and saved the correct data to be inserted into the database. The log file segment produced 4 warnings, 10 mails correctly remaining in the state tables, 1625 mail delivery attempts

[1] A mean of 27.908 lines of output per log line; each connection has 30 debugging lines, plus 21 debugging lines per result. Connections which have been cloned will have the cloned connection in their debugging output, plus another 33 debugging lines. Those numbers are approximate, and may vary ± 2. An approximate linear relationship between the number of log lines and debugging lines is: $33(connections) + 30(accepted\ \ mails) + 21(results)$.

correctly entered in the database, 0 false positives, and 0 false negatives.

Given the evidence detailed above, the author is confident that the false positive rate when reconstructing a mail delivery attempt from the 93 log files is exceedingly low, if not zero.

6.3 Summary

This chapter evaluated PLP on two criteria: efficiency, and coverage of Postfix log files. The former began by describing the mail server the log files were taken from, the computer used to generate the statistics in this chapter, and the characteristics of the 93 log files used to generate statistics and test coverage, including why performance is better when parsing the larger log files. The framework optimises the order in which rules are used when trying to recognise each log line, and the effect that optimisation has on parsing time is explored; this is followed by comparing optimal ordering with an oracle. How PLP scales as the size of the ruleset increases is addressed, followed by a description and analysis of the simplest and most effective of the optimisations, caching compiled regexes, and the efficiency evaluation concludes with an examination of where parsing time is spent: recognising log lines or processing them?

Coverage of Postfix log files is divided into two topics in this chapter: log lines correctly recognised, and mail delivery attempts correctly understood and reconstructed. The former is initially more important, because the parser must correctly recognise every log line if it is to be complete, but subsequently the latter takes precedence because correctly reconstructing the journey a mail delivery attempt takes through Postfix is the purpose of the parser. Increasing the proportion of log lines correctly recognised is relatively simple and non-intrusive: adding new rules or modifying existing rules is very easy because of the separation of rules, actions, and framework in both the architecture and PLP. Improving the understanding and reconstruction of mail delivery attempts is harder, because Postfix's behaviour must be analysed and figured out, and support for the newly understood behaviour integrated into the actions without breaking the existing parsing. Detecting

a deficiency in the parser's understanding of mail delivery attempts requires careful study of any warnings produced and the entries remaining in the state tables. Rectifying a flaw in the parser requires a deep understanding of Postfix's log files, and a working knowledge of the framework, actions, and rules; investigative work will be needed to determine the cause of the deficiency, followed by further examination of the log files to aid in developing a solution, and finally implementation, integration, and testing of the solution.

This chapter shows that it is possible to balance the conflicting goals of efficient and accurate parsing, and that one does not have to be sacrificed to achieve the other.

Chapter 7

Conclusion

Parsing Postfix log files appears at first sight to be an uncomplicated task, especially if one has previous experience in parsing log files, but it turns out to be a much more taxing project than initially expected. The variety and breadth of log lines produced by Postfix is quite surprising, because a quick survey of sample log files gives the impression that the number of distinct log line variants is quite small; this mistaken impression comes from the uneven distribution exhibited by log lines produced in normal operation, vividly illustrated in graph 6.9 [p. 138]. Given the diverse nature of Postfix log lines, and the ease with which administrators can cause new log lines to be logged (§2.3 [p. 22]), enabling users to easily extend the parser to deal with new log lines is a design imperative (§4.1 [p. 53]). Providing a tool to ease the generation of Regular Expression (regex)es from unrecognised log lines (§5.6.2 [p. 100]) should greatly help users who need to extend their ruleset to recognise previously unrecognised log lines.

This architecture's greatest strength is the ease with which parsers based on it can be adapted to deal with new requirements and inputs. Parsing a variation of an existing input is a trivial task: simply modify an existing rule or add a new rule and the task is complete. Parsing a new category of input is achieved by writing a new action and a rule for each input variant; quite often the new action will not need to interact with existing actions, but when interaction is required the framework provides the necessary facilities. The

architecture imposes very little red tape when writing new actions, allowing the implementer to focus their time and energy on correctly implementing their new action (§4.3 [p. 59]). The separation of the architecture into rules, actions, and framework (§4.1 [p. 53]) is unusual, partly because the three are separated so completely. Although parsers are often divided into separate source code files (the combination of lex & yacc [9] being a common example), the parts are usually quite internally interdependent, and will be combined by the compilation process; in contrast, Postfix Log Parser (PLP) keeps the rules and actions separate until the parser runs (§5.4 [p. 85]). This separation enables the optimisations discussed in §6.1 [p. 130], and it also allows different approaches to ruleset management, e.g. using machine learning techniques to seamlessly create or alter rules to recognise new inputs (§4.4.1 [p. 62]). The decoupling of rules from actions allows different sets of rules to be used with one set of actions, e.g. a parser might have actions to process versions one and two of a file format; by choosing the appropriate ruleset the parser will parse version one, or version two, or both versions. A general purpose framework can be written, so that writing a parser just requires writing actions and rules. The architecture makes it possible to apply commonly used programming techniques (such as object orientation, inheritance, composition, delegation, roles, modularisation, or closures) when designing and implementing a parser, simplifying the process of working within a team or developing and testing additional functionality. This architecture is ideally suited to parsing inputs that are not fully understood or do not follow a fixed grammar: the architecture warns about unrecognised inputs and errors encountered by actions, but continues parsing as best it can, allowing the developer of a new parser to decide which deficiencies are most important and require attention first, rather than being forced to fix the first error that arises.

The flow of control in this architecture is quite different from other architectures, e.g. those used for compiling a programming language. Typically, those parsers have a current state: each state has a fixed set of acceptable next states, processing is determined by the state transition that takes place, and unacceptable state transitions cause parsing to fail. This architecture is

different: the rule that recognises the input dictates the action that will be invoked. Rule conditions (§4.4.2 [p. 63]) enable stateful parsing, where the list of rules used to recognise an input is constrained by the parser's current state, but the recognising rule still dictates the action that is invoked and, whether directly or indirectly, the next state.

When writing PLP, the real difficulties arose once the parser was successfully recognising almost all of the log lines, because most of the irregularities and complications documented in §5.7 [p. 107] started to become apparent then. Adding new rules to deal with numerous infrequently occurring log line variants was a simple if tiresome task, whereas dealing with mails that were missing information or where Postfix's actions were not being correctly reconstructed was much more grueling. Trawling through log files was extremely time consuming and quite error prone, searching for something out of the ordinary that might help diagnose the problem, and eventually finding it — sometimes hundreds or even thousands of log lines away from the last occurrence of the queueid for the mail in question. Sometimes the task was not to identify the unusual log line, but to spot that a log line normally present was missing, i.e. to realise that one log line amongst thousands was absent. In all cases the evidence was used to construct a hypothesis to explain the irregularities, and that hypothesis was tested in PLP; if successful, the parser was modified to deal with the irregularities, without adversely affecting existing parsing. The complications documented in §5.7 [p. 107] are presented in the order they were solved in, and that order closely resembles the frequency in which they occur; the most frequently occurring complications dominated the warning messages produced, and so naturally they were the first complications to be dealt with.

PLP is not merely a proof of concept: it is intended to be used for parsing real-world log files from production mail servers, and the resulting data used to improve anti-spam defences. This means that efficiency is important: parsing must complete in an reasonable period of time, so that the results can be used in a timely manner. PLP's efficiency is evaluated in §6.1 [p. 130], where optimisations and the effect they have are explored.

A parser's ability to correctly parse its inputs is extremely important;

PLP's coverage of 93 log files, each containing one day's log lines, is discussed in §6.2 [p. 149]. Both its success at recognising individual log lines and its correctness in reconstructing each mail's journey through Postfix are described in detail, including the results of manually verifying that a randomly selected portion of a log file was correctly parsed. Experience implementing PLP shows that full input coverage is not difficult to achieve with this architecture, and that with enough time and effort a full understanding of the input is possible. Postfix log files would require substantial time and effort to correctly parse regardless of the architecture used; this architecture enables an iterative approach to be used [22], as is practiced in many other software engineering disciplines.

The data gathered by PLP provides the foundation for the future of this project: using machine-learning algorithms to analyse the data and optimise the set of anti-spam defences in use, followed by identifying patterns in the data that could be used to write new anti-spam techniques to recognise and reject spam rather than accepting it. The database (§5.3 [p. 76]) provides the data in a normalised form that is far easier to use as input to new or existing implementations of machine-learning algorithms than trying to adapt each algorithm to extract data directly from log files. New policy servers, written to implement new anti-spam measures, can be tested or trained by using the collected data to simulate mail delivery attempts; this would allow simple, fast, reproducible testing, without the risk of adversely affecting a production mail server. Development of PLP is finished, i.e. it correctly parses Postfix log files, and in future it will only require maintenance; however, one avenue of future development under consideration is to extend it to parse non-Postfix log lines, e.g. SpamAssassin or Amavisd-new log lines. PLP can easily be extended to do this, but it requires a method of associating the non-Postfix log lines with the existing data structures and state tables, so that all of the data for a mail delivery attempt can be stored together.

PLP provides a basis for systems administrators to monitor the effectiveness of their anti-spam measures and adapt their defences to combat new techniques used by those sending spam. PLP is a fully usable application, built to address a genuine need, rather than a proof of concept whose sole

purpose is to illustrate a new idea; it deals with the oddities and difficulties that occur in the real world, rather than a clean, idealised scenario developed to showcase the best features of a new approach.

Appendix A

Bibliography

[1] A Data Clustering Algorithm for Mining Patterns from Event Logs. Risto Vaarandi. *3rd IEEE Workshop on IP Operations and Management, 2003*.
`http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1251233`
Last checked 2009/04/21.

[2] A Plan for Spam. Paul Graham. *Hackers & Painters*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN: 0–596–00662–4.

[3] A User-Extensible and Adaptable Parser Architecture. John Tobin, Carl Vogel. *Research and Development in Intelligent Systems* volume 25, pages 191–204, 2008. Springer, 233 Spring Street, New York, NY 10013, USA. ISBN: 978–1–84882–170–5. Proceedings of AI-2008, the Twenty-eighth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence.

[4] A User-Extensible and Adaptable Parser Architecture. John Tobin, Carl Vogel. *Knowledge-Based Systems*.
DOI: 10.1016/j.knosys.2008.10.011
`http://dx.doi.org/10.1016/j.knosys.2008.10.011`
Last checked 2009/04/21.

[5] Access — Postfix SMTP Server Access Table. Wietse Venema. *Postfix documentation.*
`http://www.postfix.org/access.5.html`
Last checked 2009/04/21.

[6] Crafting an Efficient Expression. Jeffrey E. F. Friedl. *Mastering regular expressions.* O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN: 0596528124.

[7] Improved Error Reporting for Software That Uses Black-Box Components. Jungwoo Ha, Christopher J. Rossbach, Jason V. Davis, Indrajit Roy, Hany E. Ramadan, Donald E. Porter, David L. Chen, Emmett Witchel. *ACM SIGPLAN Notices* volume 42(6), pages 101–111, 2007. ACM, New York, NY, USA. ISSN 0362–1340.
DOI: 10.1145/1273442.1250747
`http://portal.acm.org/citation.cfm?id=1250747`
Last checked 2009/04/21.

[8] Instance-Based Learning Algorithms. David W. Aha, Dennis Kibler, Marc K. Albert. *Machine Learning* volume 6(1), pages 37–66, 1991. Kluwer Academic Publishers, Hingham, MA, USA. ISSN 0885–6125.
DOI: 10.1023/A:1022689900470
`http://portal.acm.org/citation.cfm?id=104717`
Last checked 2009/04/21.

[9] lex & yacc. John Levine, Tony Mason, Doug Brown. O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN: 1–56592–000–7.

[10] Local Mail Transfer Protocol. John G. Myers. *The Internet Society Requests for Comment.*
`http://www.faqs.org/rfcs/rfc2033.html`
Last checked 2009/04/21.

[11] Log Mail Analyzer: Architecture and Practical Utilizations. Maurizio Aiello, David Avanzini, Davide Chiarella, Gianluca Papaleo. *Trans-European Research and Education Networking Association.*

`http://www.terena.nl/events/tnc2006/core/getfile.php?file_id=770`
Last checked 2009/04/21.

[12] Natural Language Processing in Prolog. Gerald Gazdar, Chris Mellish. *An Introduction to Computational Linguistics*. Addison Wesley, address unknown. ISBN: 0201180537.

[13] Partial Parsing Via Finite-State Cascades. Steven Abney. *Natural Language Engineering* volume 2(4), pages 337–344, 1996. Cambridge University Press, New York, NY, USA. ISSN 1351–3249.
DOI: 10.1017/S1351324997001599
`http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=974705`
Last checked 2009/04/21.

[14] Performance in Practice of String Hashing Functions. M. V. Ramakrishna, Justin Zobel. *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*.
`http://www.cs.rmit.edu.au/~jz/fulltext/dasfaa97.ps`
Last checked 2009/04/21.

[15] Perl Regular Expressions. Perl 5 Porters. *Perl 5 Documentation*.
`http://perldoc.perl.org/perlre.html`
Last checked 2009/04/21.

[16] Postfix Architecture Overview. Wietse Venema. *Postfix documentation*.
`http://www.postfix.org/OVERVIEW.html`
Last checked 2009/04/21.

[17] Postfix Configuration Parameters. Wietse Venema. *Postfix documentation*.
`http://www.postfix.org/postconf.5.html#reject_unknown_sender_domain`
Last checked 2009/04/21.

[18] Postfix Lookup Table Overview. Wietse Venema. *Postfix documentation.*
`http://www.postfix.org/DATABASE_README.html`
Last checked 2009/04/21.

[19] Postfix Per-Client/User/etc. Access Control. Wietse Venema. *Postfix documentation.*
`http://www.postfix.org/RESTRICTION_CLASS_README.html`
Last checked 2009/04/21.

[20] Postfix SMTP Access Policy Delegation. Wietse Venema. *Postfix documentation.*
`http://www.postfix.org/SMTPD_POLICY_README.html`
Last checked 2009/04/21.

[21] Postfix SMTP Relay and Access Control. Wietse Venema. *Postfix documentation.*
`http://www.postfix.org/SMTPD_ACCESS_README.html`
Last checked 2009/04/21.

[22] Program Development by Stepwise Refinement. Niklaus Wirth. *Communications of the ACM* volume 14(4), pages 221–227, 1971. ACM, New York, NY, USA. ISSN 0001–0782.
DOI: 10.1145/362575.362577
`http://portal.acm.org/citation.cfm?doid=362575.362577`
Last checked 2009/04/21.

[23] Relaxed Online SVMs in the TREC Spam Filtering Track. D. Sculley, Gabriel M. Wachman. *Text REtrieval Conference (TREC).*
`http://trec.nist.gov/pubs/trec16/papers/tuftsu.spam.final.pdf`
Last checked 2009/04/21.

[24] RFC 1869 — SMTP Service Extensions. John Klensin, Ned Freed, Marshall T. Rose, Einar A. Stefferud, Dave Crocker. *The Internet Society Requests for Comment.*

`http://www.faqs.org/rfcs/rfc1869.html`
Last checked 2009/04/21.

[25] RFC 2821 — Simple Mail Transfer Protocol. John C. Klensin. *The Internet Society Requests for Comment.*
`http://www.faqs.org/rfcs/rfc2821.html`
Last checked 2009/04/21.

[26] RFC 3463 — Enhanced Mail System Status Codes. Gregory M. Vaudreuil. *The Internet Society Requests for Comment.*
`http://www.faqs.org/rfcs/rfc3463.html`
Last checked 2009/04/21.

[27] RFC 760 — DOD Standard Internet Protocol. Jonathan B. Postel. *The Internet Society Requests for Comment.*
`http://www.faqs.org/rfcs/rfc760.html`
Last checked 2009/04/21.

[28] RFC 821 — Simple Mail Transfer Protocol. Jonathan B. Postel. *The Internet Society Requests for Comment.*
`http://www.faqs.org/rfcs/rfc821.html`
Last checked 2009/04/21.

[29] Transition Network Grammars for Natural Language Analysis. W. A. Woods. *Communications of the ACM* volume 13(10), pages 591–606, 1970. ISSN 0001–0782.
`http://portal.acm.org/citation.cfm?id=362773`
Last checked 2009/04/21.

[30] Unix Philosophy. Wikipedia. *Wikipedia, the free encyclopedia.*
`http://en.wikipedia.org/wiki/Unix_philosophy`
Last checked 2009/04/21.

[31] Word Stemming to Enhance Spam Filtering. Shabbir Ahmed, Farzana Mithun. *First Conference on Email and Anti-Spam CEAS 2004.*
`http://www.ceas.cc/papers-2004/167.pdf`
Last checked 2009/04/21.

Appendix B

Glossary

<> <> is the sender address used for sending bounce notifications. In the Simple Mail Transfer Protocol (SMTP) conversation, all addresses are enclosed in <>, so `username@domain` is sent as <`username@domain`>; thus <> is actually an empty address, but it is always written as <> for clarity. Mail servers must not unconditionally reject mail sent from <>, or they are in violation of Request For Comments (RFC) 2821 [25].

Application Programming Interface One of the fundamental concepts when writing programs is the reuse of existing code, so that each new program does not reinvent existing wheels. An Application Programming Interface (API) defines the interface provided to the user of the existing code, and acts as a contract between the user and the provider: if the user adheres to the API the provider guarantees it will work, but is free to change the underlying implementation if the API is preserved.

AWK AWK is a general purpose programming language designed for processing text that is available as a standard utility on all Unix systems.

Backscatter When a spam sender or virus sends mail with forged sender addresses, innocent mail servers are flooded with undeliverable mail notifications from badly configured mail servers that either do not verify recipient addresses before accepting deliver attempts, or automatically reply to mail identified as spam; this is called backscatter.

Context Free Grammar A Context Free Grammar (CFG) is a grammar in which every transition or production takes the form $L \rightarrow R$, where L is a single non-terminal symbol, and R is a (possibly empty) string of terminal or non-terminal symbols. Every CFG can be recognised by a Push-Down Automata (PDA), and PDAs cannot recognise more complicated grammars, so CFGs and PDAs are equivalent in computational power.

DNS Blacklist A DNS Blacklist (DNSBL) is a simple collaborative anti-spam technique used to reject or penalise mail sent from IP addresses believed to be the source of large volumes of spam. The criteria used when deciding if an IP address should be included vary widely between DNSBLs, so before using one it is essential to check their listing policies. To use a DNSBL, Postfix makes a DNS request incorporating the IP address of the client; if the requested hostname is found the client is on the DNSBL.

Extended SMTP Extended SMTP (ESMTP) [24] provides a flexible mechanism for SMTP to be extended with new functionality, allowing new features to be tested without having to be included in the standard protocol. ESMTP is backwards compatible with SMTP: ESMTP clients and servers can interact with SMTP clients and servers without difficulty.

Finite Automata Finite Automata (FA) are computing devices that accept or recognize regular languages. They lack any form of storage, so although they can recognise languages such as $(ab)^*$ or $a^*(b^*|c)b$, they cannot count and so cannot recognise languages such as $a^n b^n$.

hash A hashing function transforms a string of characters to a number. A common usage is to maintain a data structure indexed by strings in an efficient manner. A full description is beyond the scope of this thesis, further information can be found in [14].

Joe Job A joe job describes a large amount of spam mail sent using a faked sender address with the intention of sullying the good name of the user of that address. Joe jobs are one cause of backscatter.

Local Mail Transfer Protocol Local Mail Transfer Protocol (LMTP) is a protocol derived from SMTP that removes the need for the server to maintain a mail delivery queue, instead relying on the client to maintain it. Typically the client is an Mail Transfer Agent (MTA), and the server is a delivery agent or a mail store. Full details are available in [10].

Mail Transfer Agent A Mail Transfer Agent (MTA) sends and receives mail via SMTP. Users submit mail to an MTA via their mail client (e.g. Microsoft Outlook, Thunderbird, webmail services); the sending MTA transfers the mail to the receiving MTA, which forwards the mail to another recipient, or delivers the mail to a user's mailbox or a program such as a mailing list manager.

Process Identifier There may be multiple copies of any program executing at any one time, so the program's name is not suitable as a distinguishing identifier; instead, each process is given a Process Identifier (pid) that is guaranteed to be unique for the lifetime of the process. Once the process has completed, the pid may be reused, because they are drawn from a finite pool.

Push-Down Automata A Push-Down Automata (PDA) is a computational device similar to a FA, but it can additionally use a stack to store data. A PDA can manipulate the stack during state transitions by adding or removing a single piece of data, and when determining which state transition to take, the piece of data on the top of the stack can be used in addition to the input. PDA can recognise all the languages recognised by FA, and can also recognised languages of the form $a^n b^n$.

queueid Each mail in Postfix's queue is assigned a queueid to uniquely identify it. Queueids are assigned from a limited pool, so although they are guaranteed to be unique for the lifetime of the mail they are assigned to, they may be reused later.

Regular Expression Regular Expressions are a compact, powerful method of specifying patterns that describe a set of strings. The Regular

Expression (regex) `aa*b*b` describes a set of strings, all of which start with `a`, followed by any number of `a`, then any number of `b`, and finish with `b`; the strings `ab`, `aaaaaaaab`, and `aaabbbbb` are members of that set, whereas the strings `abba`, `abcd`, and `qwerty` are not. A string can be checked against a regex to determine if the string is a member of the set of strings described by that regex.

Request For Comments The Request For Comments (RFC) series is a series of proposals defining various protocols and formats, e.g. SMTP. The name is somewhat misleading nowadays: initially the authors were asking for peer review, but these documents are now the de facto standards the Internet runs on.

Simple Mail Transfer Protocol Simple Mail Transfer Protocol (SMTP) is the protocol used for transferring mail between the sending and receiving MTA. It is a simple, human readable, plain text protocol, making it quite simple to test and debug problems with it. A detailed description of SMTP is beyond the scope of this thesis: the original protocol definition is in RFC 821 [28], later superceded by RFC 2821 [25].

Syslog Syslog is the standard logging mechanism used on Unix systems: a program sends log messages to syslog, then syslog filters and stores the messages as configured by the administrator.

Appendix C

Acronyms

API	Application Programming Interface
ATN	Augmented Transition Networks
CFG	Context Free Grammar
CSV	Comma-Separated Value
DNSBL	DNS Blacklist
ESMTP	Extended SMTP
FA	Finite Automata
LMA	Log Mail Analyzer
LMTP	Local Mail Transfer Protocol
MTA	Mail Transfer Agent
PDA	Push-Down Automata
pid	Process Identifier
PLP	Postfix Log Parser
regex	Regular Expression
RFC	Request For Comments
SLCT	Simple Logfile Clustering Tool
SMTP	Simple Mail Transfer Protocol

SPF Sender Policy Framework

SQL Structured Query Language

Appendix D

Postfix Daemons

bounce The bounce daemon is responsible for generating bounce notifications in Postfix version 2.3 and later.
`http://www.postfix.org/bounce.8.html`
Last checked 2009/02/23.

cleanup The cleanup daemon processes all incoming mail after it has been accepted and before it is delivered. It removes duplicate recipient addresses, inserts missing headers, and rewrites addresses if configured to do so.
`http://www.postfix.org/cleanup.8.html`
Last checked 2009/02/23.

lmtp Delivers mail using the Local Mail Transfer Protocol (LMTP) protocol.
`http://www.postfix.org/lmtp.8.html`
Last checked 2009/02/23.

local The Postfix component responsible for local delivery of mail (i.e. mail delivered on the server Postfix is running on); this includes alias expansion, processing of a user's `.forward` file, and delivery of the mail, whether to a user's mailbox or a program such as a mailing list manager.
`http://www.postfix.org/local.8.html`
Last checked 2009/02/23.

pickup Pickup is the daemon that deals with mail submitted locally via `postfix/postdrop`, passing the mail on to `postfix/cleanup` for further processing.
`http://www.postfix.org/pickup.8.html`
Last checked 2009/02/23.

postdrop Postdrop is used when submitting mail locally on the server: it copies its input into a newly created mail in the queue, for processing by `postfix/pickup` and subsequent delivery.
`http://www.postfix.org/postdrop.1.html`
Last checked 2009/02/23.

postsuper Used by the administrator for maintenance tasks such as deleting mails from the queue, putting mail on hold and later releasing it, and consistency checking of the mail queue.
`http://www.postfix.org/postsuper.1.html`
Last checked 2009/02/23.

qmgr Qmgr is the Postfix daemon that manages the mail queue, determining which mails will be delivered next. Qmgr groups mail based on the recipient for local mails and the destination server for remote addresses, ensuring that it achieves maximum concurrency without overwhelming destinations or wasting resources on non-responsive destinations.
`http://www.postfix.org/qmgr.8.html`
Last checked 2009/02/23.

sendmail A Postfix component that is compatible with the Sendmail mail submission program which all Unix commands that need to send mail use; it executes `postfix/postdrop` to place a new mail in the queue.
`http://www.postfix.org/sendmail.1.html`
Last checked 2009/02/23.

smtp Delivers mail using the Simple Mail Transfer Protocol (SMTP) protocol.
`http://www.postfix.org/smtp.8.html`
Last checked 2009/02/23.

smtpd The Postfix component that accepts mail via SMTP, and implements most of the anti-spam restrictions Postfix provides.
`http://www.postfix.org/smtpd.8.html`
Last checked 2009/02/23.

virtual The Postfix component responsible for delivery of mails to virtual domains. When `postfix/local` delivers mail, the destination is determined only by the portion of the email address on the left side of the `@`, whereas when `postfix/virtual` delivers mail, the destination is determined by the entire email address. For example, if the server is responsible for both the `example.org` and `example.net` domains: `postfix/local` would deliver mail for `john@example.org` and `john@example.net` to the same mailbox, whereas `postfix/virtual` would deliver mail for those addresses to different mailboxes. Virtual delivery is used where the local part of an address may be present in multiple domains, and each must be delivered to different users.
`http://www.postfix.org/virtual.8.html`
Last checked 2009/02/23.