TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Chair of Network Software

Sven Petai
001740IAPM

Detecting Anomalies in System Logs

Master's Thesis

Supervisor: Risto Vaarandi, Ph.D

Tallinn 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. The work is original and has not been submitted for any degree or diploma at any other University. I further declare that the material obtained from other sources has been duly acknowledged in the thesis.

Sven Petai ....
(signature and date)

Annotatsioon

Süsteemilogid on tihtipeale teenimatult vähekasutatud infoallikaks süsteemi üldise tervise kohta – need sisaldavad infot, mida pole üheski muus allikas. Selle informatsiooni efektiivne kasutamine võimaldab kiiremini avastada süsteemides tekkinud probleeme ja leida väiksema vaevaga nende juurpõhjuseid.

Käesolev magistritöö räägib Elion Ettevõtted AS näitel süsteemilogides leiduva informatsiooni efektiivsemast kasutuselevõtust. Peamiselt keskendub töö logidest reaalajas automaatselt anomaaliate tuvastamisele ja logides leiduva informatsiooni efektiivse visualiseerimise lahenduse loomisele.

Töö olulisimaks tulemiks on avatud lähtekoodiga logihaldusvahendi Punnsilm loomine. Punnsilm on kergekaaluline tööriist, mis on loodud välistamismeetodil (*Artificial Ignorance*) logidest huvipakkuvate sõnumite filtreerimiseks ja lahti parsitud informatsiooni edastamiseks teistele süsteemidele.

Annotation

System logs are often ignored as a realtime information source about the health of systems. This should not be so, because logs often contain information that is not available from any other source. Using this information effectively enables one to discover problems as they happen and makes it possible to find root causes of the issues with less effort.

This master's thesis discusses a path towards more effective use of the information contained in the system logs, in the example of Elion Ettevõtted AS. The main focus of this work is the creation of a solution for visualizing and monitoring that information for anomalies.

The main contribution of this thesis is the creation of an open source log management system called Punnsilm. Punnsilm is a lightweight tool for finding noteworthy events in the system logs (using the Artificial Ignorance method) and for sending parsed information to other systems.

Contents

List of Figures

List of Tables

List of listings

Glossary

API Application Programming Interface specifies how to communicate with the given software component. 25, 29

ASCII American Standard Code for Information Interchange. 17

cron Unix system program that executes scheduled commands. 19

daemon A background program that provides services to end users or other programs. 17

FTP File Transfer Protocol. 29

GIL Global Interpreter Lock is an interpreter wide lock in the Python programming language, that serializes the threads inside the interpreter instance in order to provide thread safety. 36

GUI Graphical User Interface. 30

HTTP Hypertext Transfer Protocol. 25, 29

IMAP Internet Message Access Protocol is used by e-mail clients to access e-mail on a remote server. 53

IP Internet Protocol. 23

IPTV Internet Protocol television is a system in which TV services are delivered over an IP network. 28

IPv4 Internet Protocol version 4. 17

IPv6 Internet Protocol version 6. 17

JSON JavaScript Object Notation. 26, 36

JVM Java Virtual Machine. 32

OSS Open-source software is computer software with its source code made available and licensed with a license in which the copyright holder provides the rights to study, change and distribute the software to anyone and for any purpose. 25, 30, 32

pipe The action of sending output of one program to the input of another. 18, 20

pipeline A sequence of programs that are connected by pipes. 21

POP3 Post Office Protocol is used by e-mail clients to fetch e-mail from a remote server. 53

REST Representational State Transfer. 29

SNMP Simple Network Management Protocol is a protocol for managing devices on IP networks. 45

SQL Structured Query Language. 29

STB Set-top-box is an appliance that outputs multimedia to the television set. 29

Syslog Logging protocol that allows logs from various programs to be handled centrally and in a unified manner. 23

TCP Transmission Control Protocol. 18

time serie A sequence of observations, usually recorded at constant intervals. 25

TLS Transport Layer Security. 18

UDP User Datagram Protocol. 17, 18

Unix timestamp A timestamp format often used in Unix related systems that represents time as the seconds elapsed from the Epoch at 1970-01-01T00:00:00+00:00. 25

URL Uniform Resource Locator. 55

VOD Video On Demand. 12

Chapter 1

Introduction

While participating in the development and operations of several of the largest web services in Estonia, one constant theme that the author has noticed over the years was that many problems that happened would have been detectable from the system logs, far before we actually noticed them. We did not though, since we only used logs reactively – to debug specific problems after they came to our attention through other means.

Even in this reactive mode we just used various *ad hoc* methods for finding the root cause of the problem. It became obvious that we would have gotten results much quicker if we had log visualization tools in place.

In the author's experience, relative indifference towards the great source of information that the logs are, is rather a rule than an exception. Reasons for this are discussed in detail later but in the widest sense the problem is that logs are unstructured and it requires a lot of up-front investment of time before one gets positive results.

This work describes the author's work at Elion Ettevõtted AS towards using logs more proactively, and discusses different ways of extracting information about system behavior from the logs. We start with simple methods, discuss problems with them and work our way towards more complex solutions.

1.1 Why Is Analyzing Logs Important?

There is usually a huge amount of information available in the log data which is not available from any other source. This information could be used for many different purposes from security analysis and general system health overview to marketing and usability analysis. In a commercial setting, there are often also various regulations in place that state what has to be logged and for how long it has to be kept.

In the author's specific case though, the main motivation was to get problem discovery time down. As a result, we are able to provide a better quality of service for our customers which has been main focus for our company for several years now. Reducing incident discovery time is important because it is usually a huge part of total incident response time. While we have not done any formal analysis of all the issues we have encountered, we have many specific examples where the problem discovery time accounted for more than 95 percent of the total incident response time.

One of the reasons why problem discovery takes so long, is that we usually have several different fallback mechanisms in place. Failures of a component often do not result in outright failure but rather in a degradation of the service quality. Often, the problems are also specific to the client's configuration or the services that were used (for example, problems with specific movies in the VOD service on specific platforms). Such cases that are not easily reproducible, will usually take much longer to trickle through the layers of helpdesk than problems that result in clear widespread service outages.

Another reason why it takes a long time to discover problems, is that only a small subset of the affected people will ever contact us. This is a widely known phenomenon across many industries, and usually it is said that less than 5% of unhappy customers bother to complain to the service provider. This has been confirmed by our analysis of incidents, where we have been able to count customers that encountered an issue from the logs, against the number of problem reports that we have received. In general, that percentage seems to be less than 10% even though it varies somewhat, depending on how the customer experiences the problem and how important that specific part of the service is. While people usually do not complain to the service provider, they do tell other prospective customers about such problems, which is bound to have negative impact on business.

Drastically shortening incident response time can mainly be achieved by reducing incident discovery time, which in turn can often be achieved by log analysis. This, in turn, will shorten the timeframe during which people will be able to encounter the problems. That should result in more satisfied clients.

1.2 The Problem with the Logs

There are several reasons why analyzing and monitoring logs is hard. This difficulty is the reason why it is not widely done. First of all, the log tends to be rather unstructured and one is basically dealing with the problem of parsing natural language. In the widest sense, a log message is just a string. Usually it starts with a timestamp of some kind but that can not be taken for granted. Even if the timestamp is present, it can be in a seemingly endless variety of formats that have different precision and can be ambiguous in different ways. Just as an example here are a handful of timestamps from different logs of production systems at Elion: 20140411101815.230, 20140411 123734, 1397212838, 1397212838.078857, Apr 11 12:33:25, 2014-04-11 13:35:35, 2014-04-11T13:35:35.447645+03:00.

A log message itself can be on one line or span multiple lines. If the message is spread over several lines, there might be no reliable way to detect where one message ends and another begins. Indeed, the author has encountered logs where some of the multi-line messages are intermingled, so a stack trace in the error log file might consist of several stack traces that were logged by different threads at the same time. There usually is a bit of further structure inside the content part of the log message that is imposed by the logging system, framework or the internal logging functions of the application itself. All of this depends heavily on the specific application, its configuration and the environment.

Even the structured data that are present can not be automatically trusted. For example, an analysis of supercomputer logs (Oliner and Stearley, 2007) found that 59.34% of messages that were tagged with severity `FATAL` and `FAILURE` were actually false positives.

Log messages are usually written by the developers *ad hoc* and primarily meant for themselves, so the messages might not make much sense without understanding the code that emitted them. As noted in (Allen, 2001) the developer usually writes logging calls to answer the immediate question at hand. Thus, since the developer often only wants to know *when* and *where* exactly the code fails, other questions such as *who*, *what* and *how* caused it to fail might be impossible to answer. In other cases the necessary context can be found but requires piecing it together from different moments of the same log, or from several different logs that have different granularity and format. Lack of a formal structure makes initial configuration of most generic log analyzing tools a really daunting task, which requires huge initial time investment to get up and running.

Additional complexity comes from the fact that most of the production systems are constantly evolving and consist of many different components running on many different machines. Some log message types are added, removed and modified with each version change of a software component somewhere in the system. Sometimes version changes might happen even multiple times per day. As continuous delivery becomes mainstream, the deployment frequency will increase by a large factor. For example, even back in 2011 Amazon[1] was reported to be doing deployment every 11.6 seconds during a normal business day (Jenkins, 2011).

Keeping a classical rule based log monitoring system up to date with that kind of rate of change, is a Sisyphean[2] task and, indeed, fittingly there actually is a log analysis toolkit that goes by the same name[3].

Another complication arises from the sheer amount of logs. In (Hansen and Atkins, 1993) the central log server of their system is said to log around 1MB of data per day from 12 servers, after filtering out the majority of the messages. They note that it is rather difficult and time consuming for a human to notice anything anomalous in that amount of data. Since that time, the hardware has gotten a lot better and cheaper so more data are normally logged for each operation and far more clients are served by a single server. For example, the 6 backend nodes that are serving the hot.ee portal generate about 2MB or logs *per second*. So the task of finding relevant information in the huge amount of log data, has gotten many times harder from the time when it already seemed hopeless.

1.3 Contributions of the Thesis

The objective of this thesis is to investigate different methods of extracting useful information from logs. Based on these methods, a log analysis and monitoring system is constructed for use at Elion Ettevõtted AS for IPTV and other systems. The central component of the said solution is a log management and monitoring tool

[1]Amazon: `http://amazon.com`

[2]Sisyphus: King in the Greek mythology who was damned for an eternity to roll a large boulder up a hill only to watch it roll back again.

[3]Sisyphus log-mining toolkit: `http://www.cs.sandia.gov/~jrstear/sisyphus/`

called Punnsilm, which has been open sourced under the MIT license. Punnsilm is a lightweight tool that primarily focuses on realtime parsing of structured information out of the logs, sending the information to various other systems and detection of anomalous lines using the Artificial Ignorance pattern.

1.4 Outline of the Thesis

This thesis will start with an overview of related technologies and projects that are relevant to our purpose in the chapter 2 Related Work. In the chapter 3 Solution we will describe requirements for our log analysis solution which is followed by the description of the solution itself. Chapter 4 Results gives an overview of the results and describes how widely the current solution is used. The chapter 5 Summary will summarize all of the thesis.

1.5 Acknowledgements

I would like to thank Risto Vaarandi for thorough reviews and many good ideas on how to explain things better. I am grateful to Elion Ettevõtted AS for letting me open source Punnsilm and especially to my colleague Kaspar Kalve for taking care of all the bureaucracy, configuring Punnsilm and providing countless ideas on how to make it better. I would also like to thank Marju Ignatjeva for lots of LaTeX advice and grammar reviews.

Chapter 2

Related Work

In this chapter, various existing open source log management projects are discussed. In the last couple of years, the number of open source log management tools has increased substantially, and we cannot possibly cover them all. The following selection of tools is therefore rather subjective. It is primarily meant for giving some historical perspective and background on the tools that we use in our solution or drew inspiration from.

2.1 Logging Protocols

In the telecom environment, it is very common to configure all the servers, network gear and other appliances to log into central log servers. From the security perspective, it makes it much harder for an attacker to hide his tracks after the machine has been compromised. Having all the relevant logs in one place is also convenient for debugging issues, which often involves looking at and correlating logs from several different machines. It also makes managing log access, archival and various other administrative aspects much easier.

As we are primarily using various Unix-like operating systems (Linux[1], FreeBSD[2]), we use the BSD syslog protocol for centralized logging. It is also well supported by various appliances from switches and video streamers to printers.

The BSD syslog protocol was introduced in 1984 by Eric Allman and was primarily

[1]Linux: `https://www.kernel.org/`
[2]FreeBSD: `http://www.freebsd.org`

meant for centralizing logging inside a single machine (Ranum, 2005). It was meant as a better alternative to directly writing log files from each and every program. The latter results in log files in many different places all over the system. Having all the logs go through a central syslog daemon meant that there was now a central point where one could tune granularity, retention time, rotation, archival, file naming, rate limiting and forwarding to remote systems.

The syslog protocol became the *de facto* standard on Unix systems and various networking gear. It was formalized retrospectively in RFC 3164 (Lonvick, 2001) in 2001. At that point, the standard could only describe the lowest common denominator between the existing implementations because so much software with subtly different nuances of the format and limits was already in use.

The RFC 3164 log packet consists of the following fields:

- *facility*: defines the category of the program that is logging from a predefined set of facilities.
- *severity*: determines the importance of the message from 8 predefined levels that range from *debug* to *emerg*
- *timestamp*: with format "Mmm dd hh:mm:ss" (for example *Apr 02 01:32:23*)
- *hostname*: might also be an IPv4 or IPv6 address
- *message*: starts with a tag that identifies the logging process and is followed by the contents of the message

UDP is specified as a transport protocol with maximum total packet length of 1024 bytes. Because UDP is unreliable by its very nature, some of the packets might get lost without notice. No transport layer security was specified and neither is there any protection against spoofing. Hence, any host can easily use the hostname of any other host in the hostname field and one program can easily impersonate another.

As we can see, the timestamp has only a precision of one second, which is far too imprecise for many uses nowadays. It also lacks year and timezone information, making it ambiguous. The message tag is also ambiguous and does not necessarily uniquely identify the process that is logging. RFC 3164 states that the message should only contain printable characters, preferably in the standard seven-bit ASCII charset, because there is no way of communicating charset information to the recipient. Newlines are also considered non-printable, so these are usually replaced. All of these limitations make some sense, considering the time and environment where the syslog protocol was originally created and that logging was mostly done over a local Unix domain socket.

Actual implementations of the syslog servers might, and usually do, provide various extensions to the protocol. For example, in practice, TCP and TLS support is widely available, even though RFC 3164 explicitly specifies UDP. Also, longer messages than 1024 bytes might go through.

In 2009, RFC 5424[3] was created, which specifies the version of the syslog protocol that among other enhancements, fixes all of the problems mentioned above. In addition it adds the ability to use custom structured message fields (Gerhards, 2009). Transport layer options were specified in separate RFCs, 5425[4] and 5426[5], which describe transport over TLS and UDP, respectively. In the author's experience, RFC 5424 is not widely used yet.

Even if we get the event timestamp, the hostname and the program identifier from the syslog headers, the majority of the information in the message is still in the message content field which is a free form text. This seriously hinders ones ability to write easily configurable and interoperable tools.

There have been some efforts to impose further structure on the contents of the log messages and to define a clear taxonomy. The largest of such project seems to be Common Event Expression[6] which is being implemented by project Lumberjack[7]. Both of these efforts seem to be inactive by now.

2.2 Log Parsing and Clustering

One of the *ad hoc* methods that is often used to find a reason for unexpected problems, is to filter out all the known and expected log messages in the hope that whatever remains might give hints as to what is wrong. In Unix-like systems this is usually done by piping together successive inverse `grep` commands. For example, if one wants to find all the lines in a logfile that do not contain the strings *NONCRIT*, *socket timeout* and *Permission denied*, one can achieve it with the pipeline given in Listing 1.

[3]RFC 5424: `http://tools.ietf.org/html/rfc5424`
[4]RFC 5425: `http://tools.ietf.org/html/rfc5425`
[5]RFC 5426: `http://tools.ietf.org/html/rfc5426`
[6]Common Event Expression: `http://cee.mitre.org/`
[7]Lumberjack: `https://fedorahosted.org/lumberjack/`

```
grep -v "NONCRIT" logfile | grep -v "socket timeout" | grep -v "Permission denied"
```

Listing 1: Example of a grep pipeline

The next logical step from searching logs *ad hoc* would be to write all of the known and uninteresting regular expressions down to a file, tell `grep` to read the ruleset from there and just run it periodically from the cron with the output sent to the administrator. If the remaining output turns out to be non-interesting, the administrator will just add it to the ignore ruleset file. That way one should be able to learn about new problems even before one becomes aware that there is a problem. This method is really old and is known under many names such as Active Ignorance, Artifical Ignorance (Ranum, 1997) and Sherlock Holmes Method (Collier-Brown).

The problem with this method is that the initial ruleset that one has to describe before one gets any returns for their effort, might be huge, especially if the system has many components. For example, the ruleset that we use for monitoring IPTV logs at Elion has currently over 200 rules.

This method also does not work well for web servers that are open to the internet. In these cases, one will see a lot of requests for resources that do not exist and malformed requests that attempt to exploit various security problems. These requests are done by automated attack scripts and there is little use in constantly bothering the system administrator with such log messages.

2.3 Manual Ruleset Based Approaches

There are many tools available for doing manual ruleset based monitoring and clustering. These tools range from simple shell scripts to heavyweight commercial systems that do a lot more than just matching rules.

2.3.1 Swatch

Swatch[8] which stands for Simple WATCHer is one of the oldest log analyzing tools (Hansen and Atkins, 1993) and it still sees some sporadic development. It is written in the Perl[9] programming language and is configured using simple rules. The rules consist

[8]Swatch: `http://sourceforge.net/projects/swatch/`
[9]Perl: `http://www.perl.org`

of pattern-action blocks where the pattern consists of one or more regular expressions and the action block can contain multiple actions. The most common action is sending an e-mail to the operator but one can configure it to print the matching line to the console in different colors, ring the system bell, execute a command or pipe the line to another command. Also, there is a throttle keyword which will delay the next matching of the rule that it appears in, for the given amount of time. Throttle is helpful if one wants to avoid sending out hundreds of e-mails per second, once the rule starts matching hundreds of times per second. In addition, Swatch supports adding custom actions by writing custom extension modules (Atkins, 2006).

A sample Swatch configuration file is given in Listing 2

```
watchfor /out of memory/
    echo red
    mail addresses=foo@example.com,subject=memory issues
```

Listing 2: Example of a Swatch configuration rule

This rule tells Swatch to write all the log lines that contain the phrase *"out of memory"* to the console in the red color and send an e-mail to the address *foo@example.com* with the subject *"memory issues"*.

Rules are matched sequentially in the order that they appear in the configuration file. Configuration of Artificial Ignorance can be created by defining blocks with the ignore action.

2.3.2 Logsurfer

Logsurfer is similar to Swatch but implements several additional features. It states that it is based on Swatch (Thompson) but this must be more of a philosophical heritage because it is written in C instead of Perl. The main added feature is the ability to group messages into sequences. For example, one might want to group all the normal system bootup messages into a single group and be notified only once when that group is matched. Performance is also reported to be much better than with Swatch.

2.3.3 Logcheck

Logcheck is a shell script that actually just calls the `grep` utility internally with the ruleset files. It is specifically focused on the Artificial Ignorance pattern and comes with rulesets that can handle messages occurring in the default Debian Linux[10] installation. According to the Debian Popularity Contest[11] the Logcheck package is in much wider use than Swatch even though it is less powerful. This is probably because it does not require any configuration in order to be useful, and the ruleset database is being actively maintained by the Debian project.

2.3.4 Logstash

Logstash is a generic tool for log management that is built around the pipeline concept. The configuration consists of *input*, *filter* and *output* sections each of which can contain several module configuration blocks. A large selection of modules is included and more is available from third parties. The configuration syntax also supports conditionals, so one can take different actions based on the values in the event data structure.

This structure of the configuration allows construction of log parsing and shipping pipelines that take events from a variety of sources in a variety of formats and are able to output the resulting events in different formats and protocols. In between the input and output, filters are able to modify the event in various ways. For example parsing, dropping, cloning and adding additional information to the event are supported.

Logstash comes bundled with Elasticsearch[12] which can be used for storage and fast indexed searching of the structured log data. Elasticsearch in turn comes bundled with the Kibana[13] web interface that can be used for querying and visualizing data in the Elasticsearch database.

2.4 Correlation

Event correlation is a conceptual interpretation of multiple alarms such that new meanings are assigned to these alarms (Jakobson and Weissman, 1993). Basically

[10]Debian: `http://www.debian.org/`
[11]Debian Popularity Contest: `http://popcon.debian.org/`
[12]Elasticsearch: `http://www.elasticsearch.org/`
[13]Kibana: `http://www.elasticsearch.org/overview/kibana/`

this means that some action is taken when the message arrives, based on its relation with other messages that came before it. The action might be anything – ignoring the message, creating a new synthesized message, incrementing a counter, starting a timer etc.

Many different operations can be considered to fall under the term *event correlation* and many of the log management tools implement a couple of specific examples of such operations. For example, event compression which reduces many instances of identical events into a single new event, is widely implemented in syslog servers. Another very common example of event correlation is emitting a new event when more than X similar events happen in a predetermined time interval, which is called thresholding. This approach is widely used in specialized tools. For example, there are many open source projects like SSHGuard[14] that block access to the SSH (or some other) service from the firewall, when brute force attacks are detected from the logs. Thresholding is also supported by Swatch and Logsurfer. The latter also adds support for creating and deleting dynamic rules.

In the widest sense, one can divide generic event correlators into rule based ones and the ones based on artificial intelligence algorithms (neural networks, genetic learning etc.). Rule based systems are easier to understand but require more effort to get up and running, because domain expertise and time to codify it is required. In highly dynamic environments, self learning systems might be preferred.

2.4.1 SEC

SEC[15] (Simple Event Correlator) is a lightweight platform independent OSS tool for generic event correlation that is written in Perl (Vaarandi, 2005). It is rule based and despite its relative simplicity, it can be configured to implement many event correlation operations.

A rule consists of a pattern, zero or more actions, an optional boolean context expression and possibly some other parameters. A pattern is something that should match the event and it can be a substring, regular expression, Perl subroutine or a truth value. Actions can contain various things like creating a new event, sending a message to some external system or executing a command. Rules can also manipulate contexts which basically provide in memory storage for keeping state. Boolean

[14]SSHGuard: http://www.sshguard.net/
[15]SEC: http://simple-evcorr.sourceforge.net/

context expressions can be used to place additional matching restrictions on the rule that depend on the dynamic context. Additional rule parameters can specify things such as counting thresholds and size of the correlation window.

2.5 Syslog Servers

Currently the most widely used syslog servers are syslog-ng, rsyslog and the BSD syslogd. All of these servers provide the ability to specify how and where to log each message, per logging program and many other parameters. The main difference between these tools is the amount of flexibility of the configuration, and performance. We will only cover syslog-ng here, because this syslog server is most widely used in our company and it also provides a rather interesting range of extra features.

2.5.1 Syslog-ng

syslog-ng is primarily a Syslog server but it has incorporated some extra functionality like ruleset matching, correlation, parsing structured data from messages and automatic clustering. Its original ruleset was converted from the Logcheck ruleset, so it should be able to recognize most of the common log messages that occur in a Linux system out of the box.

What makes syslog-ng interesting, is that it uses radix trees to describe the patterns instead of the regular expressions that almost all of the other tools use. It is said that a radix tree based ruleset is faster, scales better and is easier to maintain than the regular expression based solutions (Fekete, 2010). syslog-ng also has support for some correlation operations, for example contexts can be created that emit new synthetic events (Fekete, 2011). An example of this is the creation of an `sshd` login event from multiple separate log lines that each contain only part of information (for example the username might be in a different message than the IP address of the client).

Having an existing pattern database and a well performing implementation seem to make syslog-ng a rather good fit for implementing Artificial Ignorance systems. Somewhat surprisingly, the *patterndb* functionality of syslog-ng does not seem to be all that popular.

2.6 Automatic Clustering

As stated previously, a lot of time and effort is required to define and update rulesets of the log parsing tools. It would help enormously if there were tools that detected frequent patterns in the logs themselves and provided us at least with suggestions for groups, if not the actual configuration. Sadly, there are only a few such tools available.

2.6.1 SLCT and LogHound

SLCT[16] (Simple Logfile Clustering Tool) is able to detect clusters in the line based log files where each cluster represents a pattern that matches a number of lines. As such, it is useful for quickly getting an overview of what types of events the log file contains and what are the most anomalous lines that do not fit into any clusters (Vaarandi, 2008).

LogHound[17] is another project by the same author that uses a bit different method for mining clusters, called Frequent Itemset Mining. LogHound is better at finding clusters of often occurring messages at the expense of detecting anomalous lines. As such, it is a good fit for constructing an initial ruleset for the purposes of Artificial Ignorance.

The main parameter that the user has to specify for both tools is a support threshold which specifies how similar two messages have to be, in order to belong to the same cluster. There does not seem to be a particularly good method for determining a suitable value that provides good balance between over generalization and over specialization. A common method seems to be just to run the tool with a couple of different values and use the result that looks most reasonable to the domain expert.

SLCT and LogHound have been used internally by a log mining toolkit called Sisyphus. The *patterndb* of syslog-ng contains the functionality for automatically discovering clusters, which is also based on algorithms from SLCT and LogHound.

[16]SLCT: `http://ristov.users.sourceforge.net/slct/`
[17]LogHound: `http://ristov.users.sourceforge.net/loghound/`

2.6.2 Source Code Aided Methods

Another approach to finding types of message in the logs, is to start from the other end of the process by actually analyzing the functions that write the logs in the first place (Xu et al., 2009). Intuitively, it should lead to much cleaner results than mining the end result but comes with a couple of drawbacks. Depending on the environment one operates in, the most important drawback might be the requirement for the source code level access to the components one runs. With continuous increase in the usage of OSS, it is becoming less and less of a problem, but probability is high that for any largish corporate system, there are at least a couple of components that one does not have source code for. Another practical drawback is that one needs to have parsers that are capable of parsing the source code of one's tools, that might be written using different logging frameworks and are themselves written in many different programming languages. At the present such tools are not available.

2.7 Logs as Time Series

In order to make information that is extracted from the logs better accessible and analyzable, it is useful to split it into time series. As an example, one might want to count the number of HTTP requests per second, the number of logins and the mean response time to the login requests. Later such time series can be used for visualization purposes but also to spot anomalous changes.

2.7.1 Graphite

Graphite[18] is an open source real-time graphing tool and time serie database with a powerful API (Davis, 2011). While it does not have anything to do with logging specifically, it does provide a great building block for anything that wants to measure a large amount of metrics.

One can send measurements for a metric to the Graphite server without defining the metric beforehand, which makes adding new metrics really convenient. The API for sending the measurements is very simple, too. The simplest method is a line based protocol over a TCP connection where each line consists of the name, value and Unix timestamp of the metric, that are separated by spaces. The name of the

[18]Graphite: `http://launchpad.net/graphite`

metric is a string consisting of alphanumeric characters and dots. Dots are used as hierarchy separators. For example, a metric that measures HTTP responses with code 200 from the server called *publicapi1* in the IPTV cluster, might be named *dtv.publicapi1.http.code.200*.

In most installations, the data are not actually directly sent to Graphite but rather to a small proxy called Statsd[19] that performs aggregation. Statsd accepts input over UDP which means that sending of the statistics to it is a lightweight fire-and-forget event (Malpass, 2011). This decoupling is important if one wants to ensure that ones application does not get slow or break if there is a problem with the server that is gathering statistics.

Graphite has a simple but powerful HTTP based API, where the series that one is interested in and the functions that should be applied to them are given as GET parameters. There are many functions available, ranging from simple ones like *absolute()* to rather complex ones like *mostDeviant()* that returns the series having the largest standard deviation from the average. Functions can be nested and some wildcard symbols like * can be used in the names of the metrics.

For example, assuming one has a cluster of N servers and system load measurements from these machines are stored under the keys *dtv.webX.load.5min*, one can use the query parameters given in Listing 3, to get the 5 most deviant ones over the last 24 hours.

```
target=mostDeviant(5, dtv.web*.load.5min)&from=-24h
```

Listing 3: Example of a Graphite query

Besides being able to draw *ad hoc* graphs for visualization, one can also just export the data in JSON format, which makes using it from external tools very simple.

[19]Statsd: https://github.com/etsy/statsd/

Chapter 3

Solution

At the end of 2012, the operations team of the Elion's IPTV project expressed interest in analyzing their logs. The author decided to take this project because of earlier experience with log analysis for the hot.ee project. This chapter describes the environment we operate in, the requirements, the tools that were considered and the architecture and rationale behind the solution.

3.1 Requirements

The author analyzed the kind of incidents they had encountered over the years in different projects, and what would it have taken to detect these from the logs. As a result, the author came up with the following list of features that our log analysis system should have.

1. Artificial Ignorance – describing all the known messages and considering everything that doesn't match the ruleset, a noteworthy event.

2. Some of the known and described message types should be sent directly to the person(s) responsible for handling it using e-mail.

3. In some cases, the log monitoring system should be able to take direct action (restart the service when out of memory etc.)

4. Each described message type should be recorded as a time serie for easy *ad hoc* visualization.

5. Administrators need a dashboard that can be used to quickly determine the health of the system and find source of anomalies.

6. We should monitor changes in message volumes in these time series. Both rapid increase and a decrease can signal a problem.

7. In order to reduce administrative workload, we should make configuration as easy as possible. To that end, we decided to provide some test tools, investigate tools for automatic cluster detection and provide a utility for configuration visualization.

8. Sequence detection and anomalies in discrete sequences – this is mostly useful for detecting problems with clients that hang/crash while loading the application.

We also defined the following non-functional requirements:

1. The solution should be able to handle at least the current message volume of our largest systems (around 1500 messages per second) and there should be a clear path of scaling to handle larger volumes.

2. In general open source solutions are preferred because of the price, flexibility and the company wide strategy.

3.2 Environment

We will only describe environment of our IPTV project here, even though the solution described in this work is by now in use in several other projects at Elion. This is because IPTV project was the first user of this solution and because of that had the most influence on the resulting design. Other environments, that are quite different, are briefly described in the Chapter 4.

IPTV is an interesting test case, because it is a relatively large and mission critical project that serves more than 120,000 customers daily. It contains 10 clusters of machines serving different purposes, and runs 2 major production versions of our IPTV code in parallel. We can be relatively confident that whatever works for IPTV performance and configuration flexibility wise can be transferable to our other projects that are more homogeneous and less loaded.

The user interface of the Elion IPTV solution that clients see on their TV screens, is actually a web application written in Javascript. The browser is running in full screen mode inside the STB. Applications communicate with our servers using a REST HTTP API. So on the server side the majority of the logs are produced by the web servers (Nginx[1] in our case) and PHP[2] web applications running on top of it. There are also various other components in the system like proprietary appliances, FTP servers, cache servers, SQL servers and others.

Several of our IPTV clusters share large parts of the code base, but also have slight differences. This suggests that the solution should allow for an easy reuse of various subsets of the ruleset between different log sources.

IPTV project had a central syslog server in place and everything was already configured to log there. In some of the other environments, where we moved later, such preconditions were not met and took months to implement.

3.3 Selection of the Solution

This section discusses the reasoning behind the selection of the tools that we decided to use. First an overview is given of the relevant commercial tools, and reasons behind not using any of these. The author will then discuss the open source offerings and explain the reasoning for writing our own tool, in addition to using several open source ones.

3.3.1 Commercial Options

One of the best known proprietary log handling tools is Splunk[3] which is widely used by enterprises and has impressive range of features. It provides functionality for collecting, exploring and analyzing logs and other machine generated data. It provides log indexing, search, visualization, report generation, event correlation and lots of other functionality. Splunk is often criticized for its price and per log volume licensing policy which encourages one to log less. It does, though, provide a well integrated set of functionality. It would take a lot of integration and configuration effort to achieve a similar result with various open source tools.

[1]Nginx: `http://nginx.org/`
[2]PHP: `http://www.php.net`
[3]Splunk: `http://www.splunk.com`

There are some cheaper cloud based alternatives like Loggly[4] that are becoming popular, but since logs contain a lot of sensitive data, sending it to an external party would have been a violation of our security policies.

Our log analysis effort was largely a skunkworks "project" without budget or management involvement. It would have been theoretically possible to secure funding for commercial tools, but we felt it would have taken far too much work and probably more time than we spent on implementing what we needed. We saw the need for a rather small and well defined set of functionality which is a tiny subset of the functionality that the large integrated commercial tools like Splunk provide.

3.3.2 Open Source Tools

It was evident that in the last couple of years, something of a Cambrian explosion[5] has happened in the world of OSS log tools. There are layers upon layers of tools popping up for just about any function one can think of. At every layer of the stack, one can choose between several powerful alternatives that are actively developed and have different strengths. Besides large projects such as Kibana[6], Logstash[7], Graylog2[8], ELSA[9] and Fluentd[10], there are also host of smaller projects. With some integration, these tools can be combined together in various ways to create tailor made solutions. As an example, many people seem to not like how Graphite UI looks like, so there are tens of different alternative GUI layer projects for Graphite such as Grafana[11] and Graphene [12]. Having the ability choose components from such a wide pool of alternatives and being able to evolve our system as the needs change, directed our choice towards open source solutions.

The author decided to use Graphite in our solution for storing information from logs as time series and providing visualization. That choice was primarily motivated by the ability to accept new metrics without the need to pre-define anything, and also because of its great and easily extendable query language. PyStatsd was selected to be used as a statsd server. As discussed in Chapter 2.7.1 statsd is an aggregating

[4]Loggly: https://www.loggly.com/

[5]Cambrian explosion: relatively rapid appearance of wide variety of animals around 542 million years ago

[6]Kibana: http://www.elasticsearch.org/overview/kibana/

[7]Logstash: http://logstash.net/

[8]Graylog2: http://graylog2.org/

[9]ELSA: https://code.google.com/p/enterprise-log-search-and-archive/

[10]Fluentd: http://fluentd.org/

[11]Grafana: http://grafana.org/

[12]Graphene: http://jondot.github.io/graphene/

proxy that is often used in front of Graphite to make data gathering more efficient and robust. We decided to use PyStatsd instead of the original StatsD server, because the original statsd is written in Node.js, which would have added an rather inconvenient extra dependency to the system. We also decided to use Grafana on top of Graphite for more pleasing UI.

With the Graphite selection in place we had to find a tool for parsing logs and sending data to it. We also had to find a tool for the Artificial Ignorance requirement and something for monitoring the time series in Graphite.

After analyzing some of the tools described in Chapter 2, the author came to a conclusion that none of the existing tools is a perfect fit for our requirements. For example, Logstash is a great log shipping tool with huge amount of modules, but it is somewhat heavyweight because it runs on top of JVM and is not built for implementing Artificial Ignorance pattern. Other tools, such as LogCheck, are good at Artificial Ignorance but lack the ability to do real time analysis or log shipping. The author saw a lot of overlap between the configuration of these two functionalities, so just using different tools for these functions would have led to repeating the logically almost identical configuration in several places using different syntax. The author already had written lightweight Artificial Ignorance tool in 2009 for the hot.ee project, so the decision was made to extend that tool with the log shipping functionality that we required.

In addition we also use open source Graphviz[13] graph visualization software in our solution for visualizing configurations of our tool.

3.3.3 Punnsilm

The log management tool that the author wrote is called Punnsilm. It is primarily built as a lightweight tool for realtime Artificial Ignorance matching but also provides some log shipping functionality. Punnsilm is primarily designed with our IPTV solution in mind and our purpose was to make the configuration for that environment as easily maintainable and repetition free as possible. This focus has resulted in providing syntax for heavy reuse of different subsets of the rule chain for multiple log sources and the decision to use regular expression group names for extracting metrics for Graphite. That approach allows us to extract large amount of time series from the logs with very little configuration.

[13]Graphviz: http://www.graphviz.org/

Punnsilm also contains a tool for monitoring time , called *graphite-analyzer*. There are several existing OSS tools that provide monitoring and alerting support on top of Graphite like Seyren[14] and Rearview[15]. We did not want to use these since they were far too heavyweight solutions for our needs, and would have introduced several new programming languages, dependencies and configuration languages into the mix.

The author decided not to implement any correlation functionality inside our own tool, and instead to provide a means to integrate our tool with SEC. We are not currently using SEC yet in our solution though, primarily because we have not yet found any good examples of problems that would have needed event correlation support for detection.

3.3.4 Choosing the Language

There is a policy in Elion, that all the new software should be written either in Java or Python, unless there is a really good argument for doing it in some other language.

The author chose Python for our tool because:

1. It should be very easy to extend the tool. In the author's experience, this is far easier in Python than in Java because there is a lot less boilerplate code and tools needed to be productive. All one needs to write a plugin for Punnsilm is a working text editor.

2. Most of our system administrators already know Python but only few are able to write Java. Since the primary users of this tool will be operations people, it is very important to make it as easy as possible to extend and configure for them. Otherwise, the tool will just languish.

3. Python is increasingly taking over the scientific computing world (Yarkoni, 2013), which means that there are many high performance data crunching and machine learning libraries available. Since some of our future goals are clearly machine learning related, having easy access to such a great resource is a plus.

4. Python can easily be interfaced with almost any other language, while interfacing Java with anything that does not run on top of JVM is rather hard.

[14]Seyren: `https://github.com/scobal/seyren`
[15]Rearview: `https://github.com/livingsocial/rearview`

In addition to choosing Python as the language, the author also had to choose which versions of it to support and how to do it. Python 2.X and 3.X series have some incompatible syntax changes between them, which makes supporting both at the same time non-trivial task. Even though Python 3.0 was released in 2008, the 2.X series is, at the time of writing still far more popular, because of the huge amount of libraries and proprietary code that is not yet Python 3 compatible. The author decided to use Python 3.2 as a primary target, and employ workarounds in code that make it backward compatible with Python 2.7.

3.4 The Architecture of Punnsilm

Punnsilm consists of various nodes that are connected in a graph like manner. Configuration for each node consists of a name, type, parameters and optionally an empty output list that holds the names of nodes to send output to. Configuration of the Punnsilm consists of a list of such node configurations which together logically describe a graph. There are three basic node types: input-, output- and intermediate nodes.

Input nodes are for example log file monitors, socket listeners and syslog listeners. Intermediate nodes do filtering and rewriting of the messages. The most used intermediate node in our configurations is the *rx_grouper* which holds a set of regular expression groups that can all have different output lists. Output nodes have an effect on the world outside the graph like sending an e-mail, writing out logs, printing messages or executing external commands.

Usually the configuration of Punnsilm is expected to form a directed acyclic graph, but there are no safeguards in place to ensure that this is the case, so one can create configurations that lead to infinite loops. Indeed one can think of configurations were cycles make sense. For example, a normalizing node might want to feed a message back to the parser node before it, that is able to parse it differently after normalization step. Additionally one can also introduce cycles through external components for example by writing to syslog that is also read from one of the inputs.

Punnsilm has a global state manager that each node can use to write down any state information that it wants to keep over the program restart. This is useful for storing things like how far parser has gotten in the log so that it would not start from the beginning, when was the last e-mail sent and so on.

The architecture is highly modular and most of the node types are actually implemented as modules. The core only contains base classes, a state manager and functions for reading configuration and connecting the graph together.

Users can extend the tool by writing custom nodes that implement a simple interface and placing the module inside the modules directory of Punnsilm.

3.4.1 Modules

Table 3.1 describes the modules that are available in the core distribution of Punnsilm.

type	function
`file_monitor`	Provides support for reading log files. Has support for name expansions and handles rotation.
`syslog_file_monitor`	Provides parsers for several of the most common syslog log file formats.
`syslog_input`	Listens on a TCP or UDP socket and is able to handle Syslog protocol.
`graphite_input`	Time serie monitor that periodically checks Graphite dashboard values.
`rx_grouper`	Parses and groups messages using regular expressions.
`rewriter`	Allows rewriting of the message contents.
`console_output`	Prints messages to the output streams. Supports different output streams and colors.
`log_output`	Outputs messages to the log.
`pipe_output`	Writes output into a named pipe or command over a Unix pipeline.
`statsd_output`	Sends messages to the Statsd daemon (an aggregating proxy for Graphite).
`smtp_output`	Sends out e-mail.
`http_output`	Allows sending messages over HTTP protocol.
`mariadb_output`	Executes SQL queries in MariaDB. Can be used for updating user login timestamps, writing log to the SQL database etc.

Table 3.1: The modules of Punnsilm

We also have a couple of modules that are not public and are required for integration with custom systems that we use.

An in-depth documentation about the functionality and configuration options of these nodes is available on the projects homepage[16]. Various configuration examples are available in the `test` directory of the project.

In order to write a custom module for Punnsilm one only has to extend the `core.PunnsilmNode` class and optionally implement a couple of methods. Once the Python file containing the new module is placed into the modules directory of Punnsilm it will be automatically picked up. Useful modules can be written in as little as 5 lines of code, which lowers the barrier of entry for people who are not primarily programmers. Indeed at least one such custom module has been written by the user with minimal guidance from the author.

3.4.2 Performance

Using a scripting language like Python, for something that demands relatively high performance, might at first seem a bit counter intuitive. In practice this did not worry us since tools like SEC and Logstash (written in Perl and Ruby respectively) have already proven that handling load similar to ours is possible, with software that is written in a high level scripting language. The productivity gains that resulted from using a very high level language (as compared to C or C++) were deemed far more important than getting the best possible performance.

The author's goal has also been a horizontally scalable architecture rather than one which squeezes maximum throughput out from a single machine. The graph structure of Punnsilm lends itself well to this goal, since there is a very clear communication boundary between the nodes. This allows one to write node types in different languages and possibly, run these even on different machines. Currently, we have not actually written any such nodes but it was an important design consideration that influenced the design because it would have been hard to add afterwards.

A reasonable effort was also invested in getting good performance on a single machine. Each input node is started in a separate thread which gives us the ability to use multiple CPU cores better. In the reference implementation of Python (*CPython*) there is a Global Interpreter Lock (GIL) which serializes threads inside each interpreter instance, so for CPU bound loads, using multiple threads does not actually help performance wise. Indeed, it might actually have a negative impact on the performance. In our specific case though, there are several reasons why using threads *does* give a performance boost. First of all, in most Punnsilm configurations the majority of the

[16]Punnsilm: http://bitbucket.org/hadara/punnsilm/

time is spent parsing strings and usually, it is done with regular expressions. For matching regular expression we use the `regex`[17] library that is a drop-in replacement for the `re` library that comes with Python. The `regex` library has various performance enhancements and drops the GIL while matching, so that the other threads can run meanwhile. So in our specific case, we are actually able to get a performance boost from using multiple threads in Python.

Python also has very good support for integration with other languages, which makes it easy to write the few performance critical functions in a language that has better performance. A popular and simple method for doing that is through Cython[18] which is a superset of Python that compiles to C code. This approach allows one to keep the readability of Python, while gaining the performance that can be had in C. There are currently no Cython based modules in the core Punnsilm distribution because there has been no clear need for that, and the author did not want to introduce an extra dependency for Punnsilm. The author did perform some experiments though, to verify that such modules indeed work and give the expected boost in performance.

3.5 Configuration

The configuration files of Punnsilm are expected to be valid Python source files. In general, the configuration should be kept as simple as possible and should ideally contain nothing besides variable and node definitions. Nothing stops one from using loops and conditionals in the configuration but it has to be kept in mind that these will be evaluated at startup time, and will make configuration hard to follow for non-programmers.

Originally, we started out with using JSON for configuration files because of its great portability. Its lack of syntax for comments and other small inconveniences were the reason for the switch to Python.

Configuration files can include other configuration files which allows one to better structure the configuration and to reuse some of the sub configuration files between different configurations.

The simplest Artificial Ignorance configuration for Punnsilm contains only 3 nodes and is visualized in Figure 3.1.

[17]regex: `https://pypi.python.org/pypi/regex`
[18]Cython: `http://cython.org/`

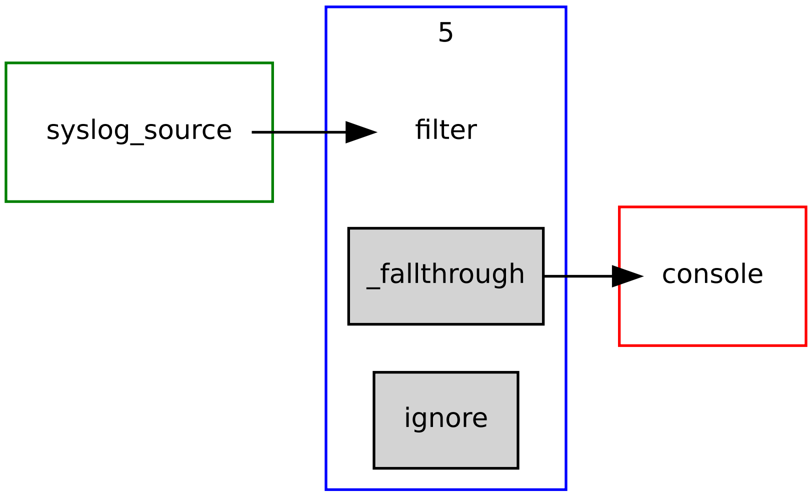


Figure 3.1: Minimal configuration graph implementing the Artificial Ignorance pattern.

In Figure 3.1 and a couple of subsequent ones, the green boxes are used to represent input nodes, the blue boxes show intermediate *rx_grouper* nodes and the red ones are output nodes. The number in the blue boxes indicates how many regular expressions that group contains. The *rx_grouper* node named *filter* contains five regular expressions that match the known messages. It also contains the special group named *_fallthrough* which is passed all the messages that do not match any of the regular expressions. The *_fallthrough* group just passes all the messages to the output node that prints these out. The full configuration, the input logfile and the output for this configuration are given in Appendix A.

In order to show the power of Punnsilm, Figure 3.2 presents a bit more complex example involving multiple log sources that share some of the intermediate nodes and have several different outputs. Reusing some internal paths after removing the differences, makes it possible to write shorter configurations for various clusters that are at some level different but still share a lot of code. Specifically, this example shows two input nodes called *syslog_source_web* and *syslog_source_sql* that monitor web servers and SQL servers respectively. These inputs are connected to the *rx_grouper* nodes that handle messages which are specific to the given input. The webserver specific parser node called *web_filter* is configured to send some of its output to the *statsd* output module. Both *web_filter* and *sql_filter* send everything that does not match their rules to the *sys_filter* node which contains rules for parsing system level messages that are common between the servers (for example notices about administrator logins). Everything that is not matched by *sys_filter* is printed out to the console and sent out to the administrators over e-mail.

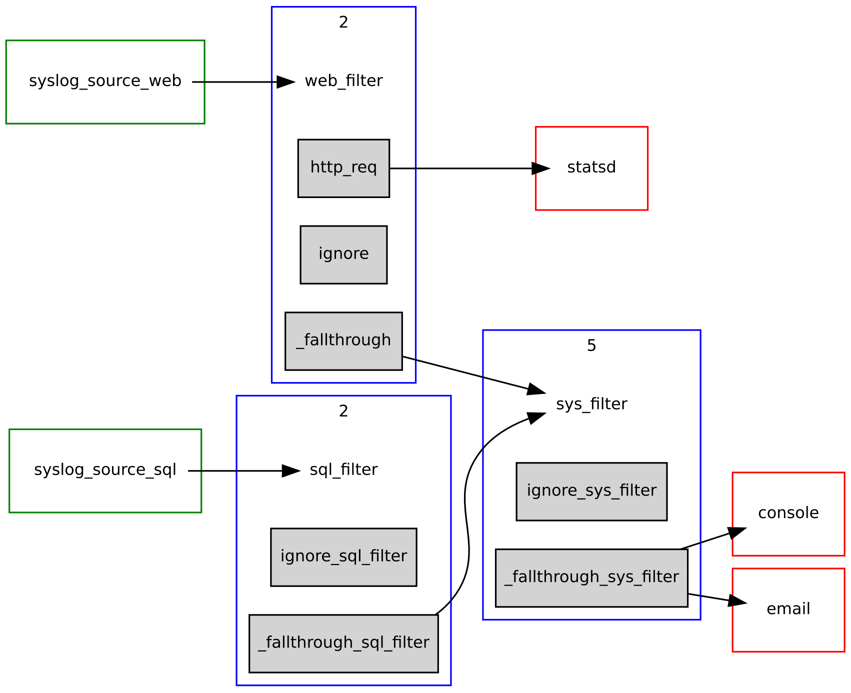


Figure 3.2: Configuration graph implementing the Artificial Ignorance pattern.

The configuration that we use to handle IPTV logs is much more complex and uses many layers of intermediate nodes that are shared between data coming from different inputs, and has many different outputs. Figure 3.3 shows the high level connectivity between the nodes of a specific subset of the IPTV configuration.

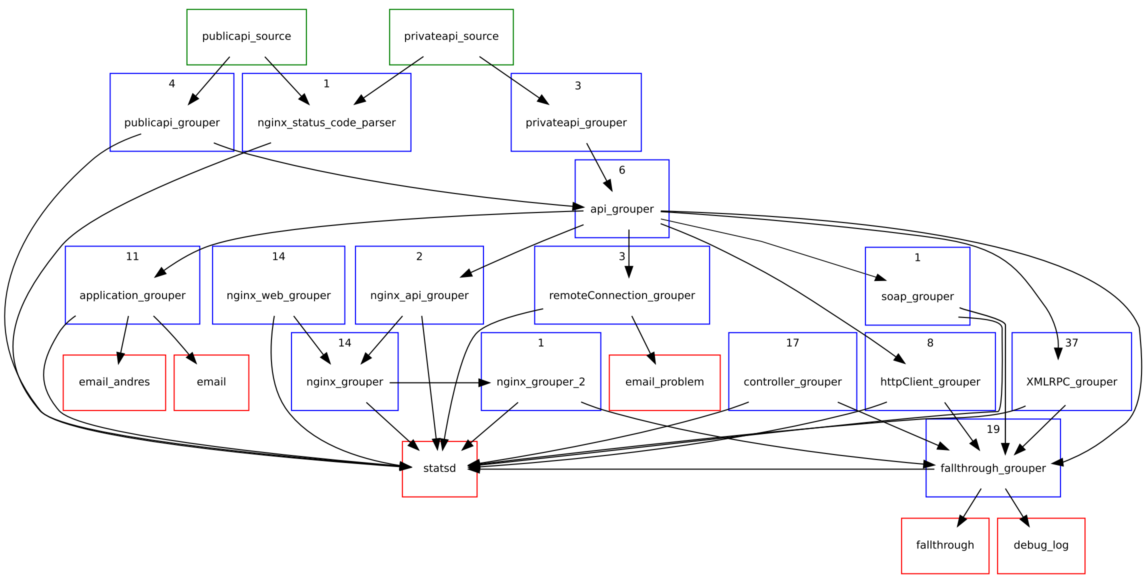


Figure 3.3: Subset of the production configuration of Punnsilm in the IPTV environment

All the configuration visualizations in this section were created using configuration visualization tool that is distributed with Punnsilm under the *tools* directory.

In large part, the configuration of Punnsilm consists of regular expressions in the parameters of various *rx_grouper* nodes that match and parse messages. There are two common criticisms that are often levied against using regular expressions for that purpose.

The first one is that regular expressions are hard to write and maintain. Listing 4 shows the example given in (Fekete, 2010) to illustrate this.

```
A log message from an OpenSSH server:
    Accepted password for joe from 10.50.0.247 port 42156 ssh2
A regular expression that describes this log message and its variants:
    Accepted \
        (gssapi(-with-mic|-keyex)?|rsa|dsa|password|publickey|keyboard-interactive/pam) \
        for [^[:space:]]+ from [^[:space:]]+ port [0-9]+( (ssh|ssh2))?
An equivalent pattern for the syslog-ng pattern database:
    Accepted @QSTRING:auth_method: @ for @QSTRING:username: @ from \
        @QSTRING:client_addr: @ port @NUMBER:port:@ @QSTRING:protocol_version: @
```

Listing 4: syslog-ng configuration example

While it is hard to argue with the point that the syntax used in the syslog-ng part of the example is indeed easier to read and write than the variant with regular expressions, it is also a bit of a false dichotomy. If the syntax of a tool allows one to define variables holding sub regular expressions and use these shared variables in the construction of regular expressions, the end result does not have to look much different from the syslog-ng example. Such a configuration example for Punnsilm that uses variables for sub patterns is available in Appendix B.

The second common criticism against using regular expressions is that they are slow. Indeed, using radix trees like syslog-ng does, or a parser specification language should be faster, and the difference might easily be by a factor or more. On the other hand, many people are already familiar with regular expression syntax and there are lots of high quality learning materials and tools available that help one to debug and visualize expressions[19]. The performance of regular expressions is also tied to specific constructs used and approaches taken in the implementation of the regular expression library. In the author's experience, performance has not been much of a problem unless advanced concepts like *lookarounds* are used. So in our use case, familiarity of the syntax and ease of implementation were considered to be more important than getting extra performance.

It is also worth noting that subsequent *rx_grouper* nodes can match rules against specific fields of the message that were already parsed by upstream nodes. This method allows one to write shorter and simpler expressions in the lower layers, which, besides being easier to maintain, should also perform better.

3.6 Logs as Time Series

Extracting a lot of time serie data from the log was one of the main goals of our solution from the beginning, so the author tried to require as little configuration as possible in order to achieve it. Punnsilm does this by heavily exploiting named regular expression groups and having the *statsd_output* module interpret various forms of group names as having certain implicit meanings. The actual mechanism behind the scenes is that *rx_grouper* sets all the matched named regular expression groups as attributes of the message that is passed to the next graph node. The *statsd_output* module iterates over the message fields and sends out measurements for the fields that match its naming conventions. So the connection between *rx_grouper* and *statsd_output* is logical rather than technical.

In Listing 5 an example is given of how this looks like in practice.

[19]Debuggex: https://www.debuggex.com/

```
Dec 20 13:21:09 publicapi8 nginx: 127.26.108.212 -
10.219.102.129 - - 20/Dec/2013:13:21:09 +0200
"GET /api/index/et/help HTTP/1.1" 200 787 0.023 0.021
"http://static.example.com/html5/"
"Mozilla/5.0 (Linux) AppleWebKit/534.51 (KHTML, like Gecko) Safari/534.51" .
```

Listing 5: Example of a log line

If we pass the message shown in Listing 5 through the configuration described in Appendix B, the four measurements shown in Listing 6 will be sent to the StatsD server.

```
Counter:test.publicapi8.http.code.200: 1
Counter:test.publicapi8.controller.help: 1
Timer:test.publicapi8.group1: 23.00000000ms
Timer:test.publicapi8.group1.help: 21.00000000ms
```

Listing 6: Example of StatsD messages

The first two measurements are counters that count the total number of HTTP responses with the status code of 200 and the number of requests against the controller called *help*. The last components of the both time serie names are extracted from the log entry itself and the name of the regular expression group is used in between the hostname and the dynamic part. The last two measurements are of the timer type which is also determined from the name of the regular expression group that extracted them.

For measurements of the counter type, StatsD will just sum the measurements for each time serie each second, and send the result to Graphite. For timers it will calculate *lower*, *mean*, *upper* and *upper_90* series for each time interval and send these to Graphite. The first three of these should be self evident and the last one is the upper value for the bottom 90th percentile. So for this single line, we will end up with measurements in 10 different time series and this was achieved using a single (admittedly lengthy) regular expression.

To give a better idea how this information can be used for *ad hoc* problem solving, we will now present an overview of a realistic debug session. Let us say the system administrator gets some reports that there is an anomalously high number of complaints from the customers. Not much additional information is available because the helpdesk has been unable to notice any common pattern in the complaints, other than that they are all related to IPTV and an older version of the middleware. A good starting point for analyzing the issue is to open up the Graphite dashboard for the

IPTV services. There, as shown in Figure 3.4, the following graph should immediately catch an eye because of all the red that is usually used to indicate trouble:

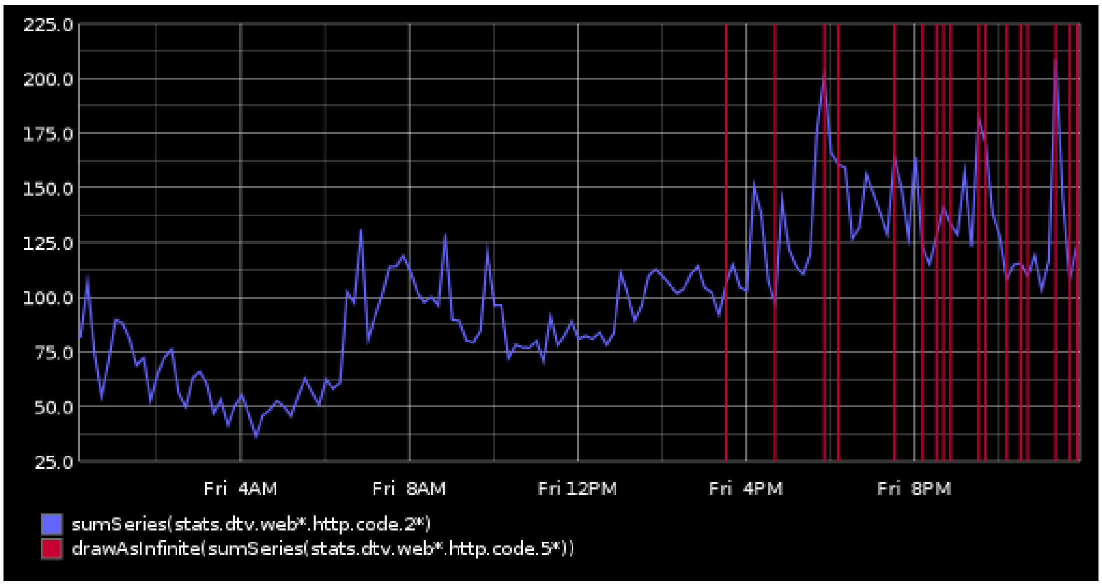


Figure 3.4: Error situation in one of the webserver clusters

As we can see from the legend, the red lines indicate HTTP 5XX reponses which is a family of HTTP response codes that indicate all kinds of temporary errors. Knowing that there are many nodes in that particular cluster, the next logical question to ask is if these errors come from all the cluster nodes or only some of them. This question can be easily answered with the Graphite query given in Listing 7, which selects the top 3 hosts sorted by the 1 minute count of HTTP 500 errors.

```
highestAverage(hitcount(stats.dtv.web*.http.code.500, "1min"), 3)
```

Listing 7: Graphite query for finding hosts with most HTTP 500 errors

This query will return the graph given in Figure 3.5.

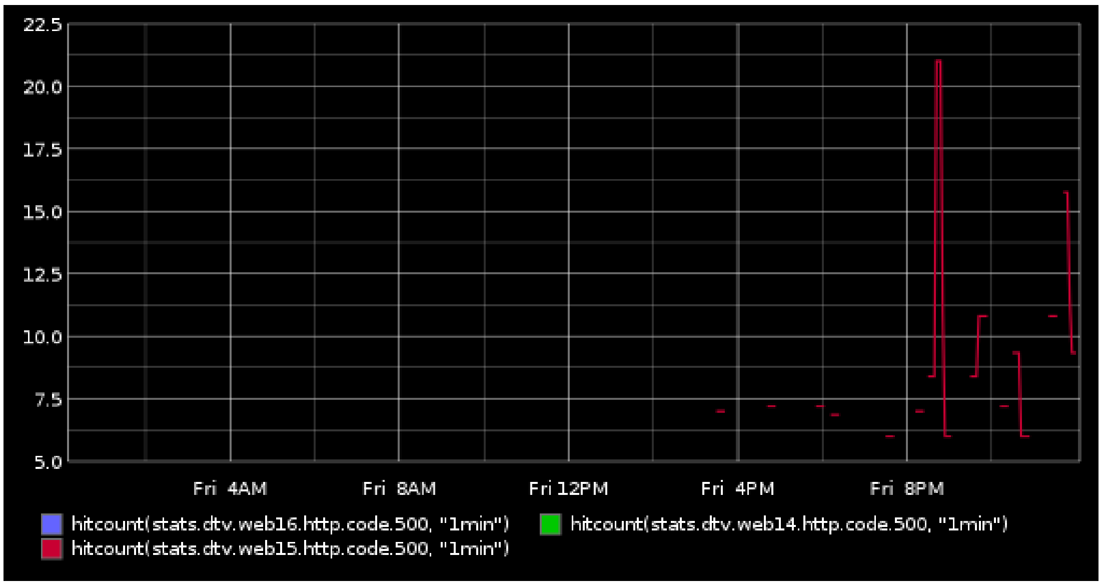


Figure 3.5: Top 3 nodes of web cluster sorted by the number of HTTP 500 errors

From this picture, it is clear that only a single server is giving errors for some reason. In this particular case the administrator remembered right away that this node had been added to the cluster the same day. After checking the error messages from this particular node, it became quickly clear that a broken firewall configuration was the cause of the problem.

We have defined a number of dashboards that contain similar graphs representing different aspects of the IPTV system and these are in day-to-day use for debugging problems.

3.7 Monitoring Time Series

Having hundreds of graphs is nice but human attention is limited so we clearly needed an automated tool that would notify us if there is a problem visible on some of the important graphs. The following section discusses how to actually define what abnormal values are, and how the author implemented the defining of abnormal values in our time serie monitoring tool *graphite-monitor*.

3.7.1 graphite-monitor

The *graphite-monitor* tool used to be separate from the main Punnsilm codebase but at some point it became obvious that some of the output modules of Punnsilm, like the e-mail output, would be useful for that tool too. In order to avoid repeating the same configuration, the author merged the tools, so that *graphite-monitor* is nowadays another Punnsilm setup that gets its input from the *graphite_input* module.

We also used to have a specialized configuration syntax for defining what to monitor and what the thresholds are. On the one hand, it allowed extreme flexibility on, making rather complicated rules, on the other hand author felt from the beginning, that having yet another configuration language was too cumbersome and would probably seriously hinder the acceptance of the tool. It seemed desirable to make it possible for every operations and development employee to add new time series into monitoring without having to request it from administrators, or having to learn the configuration language of yet another tool. Lowering the barrier of entry in such a way should greatly increase the probability that new things will get added on top of the initial configuration by the author.

Our solution was to gather all the monitored graphs into special dashboards, which *graphite_input* is configured to periodically monitor. Each graph is expected to contain a serie called *current* and at least one of the series *upper* and *lower*. The monitoring tool does not care how these series are defined and just checks if the value of *current* is between *upper* and *lower*. Because Graphite is able to output the graph data as JSON, fetching and parsing of the time serie data turned out to be surprisingly easy.

Internally, the module keeps track of state and only emits an event to its outputs when the alarm status for the graph changes. State is also kept over restarts of the monitoring tool similarly to the *file_input* module.

In the following sections we will discuss some strategies for defining such graphs.

3.7.2 Constant Thresholds

In some cases, defining a simple constant threshold is enough. For example, one might say that if the median response time for a specific request type takes over 100ms, the system should raise a warning, and once the response time goes above 200ms, an alarm should be raised. Constant thresholds work especially well if one is dealing with a well known finite resource like the amount of RAM, disk space or network throughput limits. These specific examples are usually available through better structured means than the log (for example SNMP) and are monitored by other tools, such as Opsview[20]. There are, though, several examples where the information about resource usage is only available from the log and in these cases Punnsilm is indeed the simplest method of monitoring them. Practical examples include licensed bandwidth usage in custom appliances and stack sizes of specific application server instances.

Constant thresholds are also suitable for monitoring values that should, under normal circumstances, be zero. A real life example is the number of requests per second for the URLs that serve the failover environment of the IPTV middleware.

3.7.3 Dynamic Thresholds

In many cases, there is no clear static value that could be used as a threshold.

Even monitoring some time series that measure errors is often not as straightforward as just raising an alarm if one sees an error. As an example, it is normal to constantly see a small number of HTTP requests failing with connection reset errors because of various problems on the client side like power failures, software crashes and various connectivity issues. Under normal circumstances, the number of these alarms per time interval should be proportional to the load of the application servers. Sudden change in the amount of connection errors might, on the other hand, be noteworthy and might well signal network trouble on our side.

If the service at hand is primarily used from a single time zone, like almost all of our services are, there are usually clear periodic components present in the time series that measure aspects of that service. For example, if we want to monitor the number of HTTP requests in general, or for a specific URL type in particular, there are normal variances during day and night as users go to sleep, wake up, leave for work and get back home.

In the aforementioned case with connection resets, there are far less of these during night time. If the service is primarily providing entertainment content that people use at home, the weekends will also see far higher load than the workdays. The inverse is true for business oriented services that are primarily used during workdays.

One such example of a service that is mainly used during work days and has clear

[20]Opsview: `http://www.opsview.com/`

periodic components, is shown in Figure 3.6. Besides the uniformly lower load during the weekends, one can also see that the load drops off on Friday evenings as some people turn off their computers before leaving for the weekend.

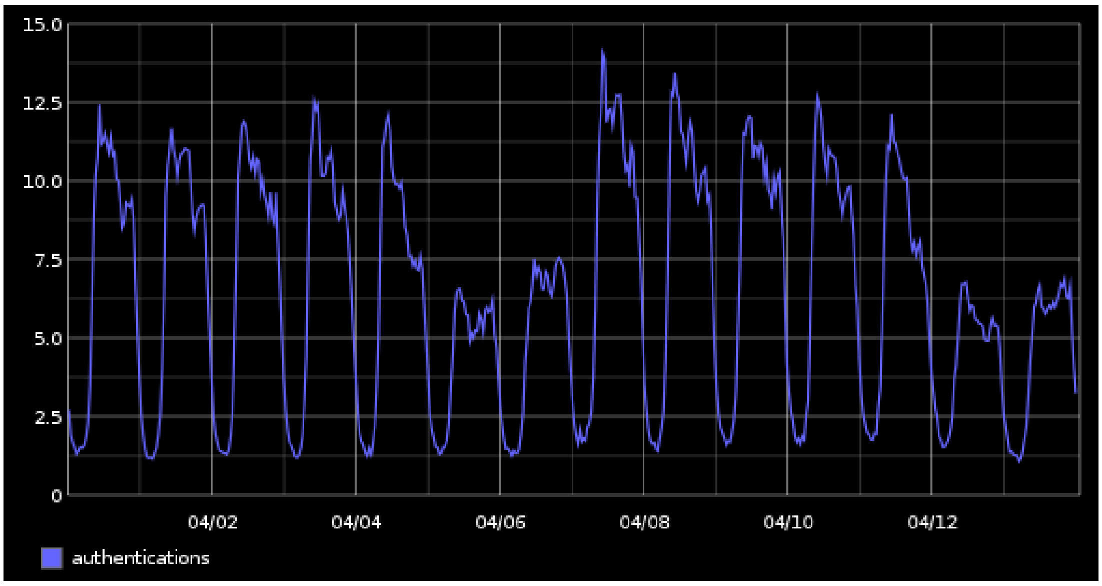


Figure 3.6: Authentication events per second showing periodic components

There might be even longer periodic components present and there might also be an underlying slower trend in some direction, as a service gets more or less popular.

In some cases, a good indicator of trouble is the speed of change itself rather than the actual value. Yet in other cases, rapid change at certain times is directly caused by the implementation of the service in question and would result in false alarms if historical context was not considered, too. Such a graph is shown in Figure 3.7 which shows the load of a service that is automatically refreshed each night.

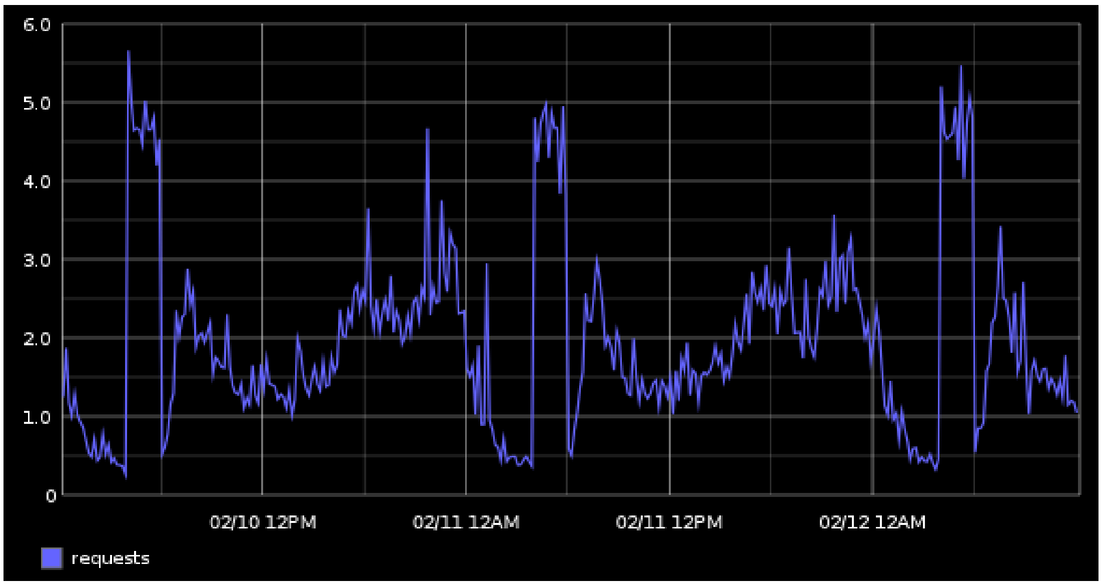


Figure 3.7: Sudden periodic nightly peak in load that is caused by service logic

In general, a good anomaly detection algorithm should be able to detect anomalies while being able to account for various periodic components, trends and sudden jumps that are periodic.

3.7.4 Moving Average

A simple approach for monitoring a time serie for sudden changes is to compare the current value against the moving average (or median) from a larger period (for example the last 5 minutes). If the current value is more than, for example, 200 percent larger or smaller than the moving average, it could possibly indicate a problem.

The moving average is defined in Equation 3.1, where k is the size of the moving window.

$$s_t = \frac{1}{k} \sum_{i=0}^{k-1} x_{t-i} \tag{3.1}$$

This method allows one to avoid triggering alarms for the slow ramp ups that are usually normal, while still triggering an alarm for rapid changes that are often caused by something breaking. An example graph of moving average and moving median of a response time during an incident is given in Figure 3.8.

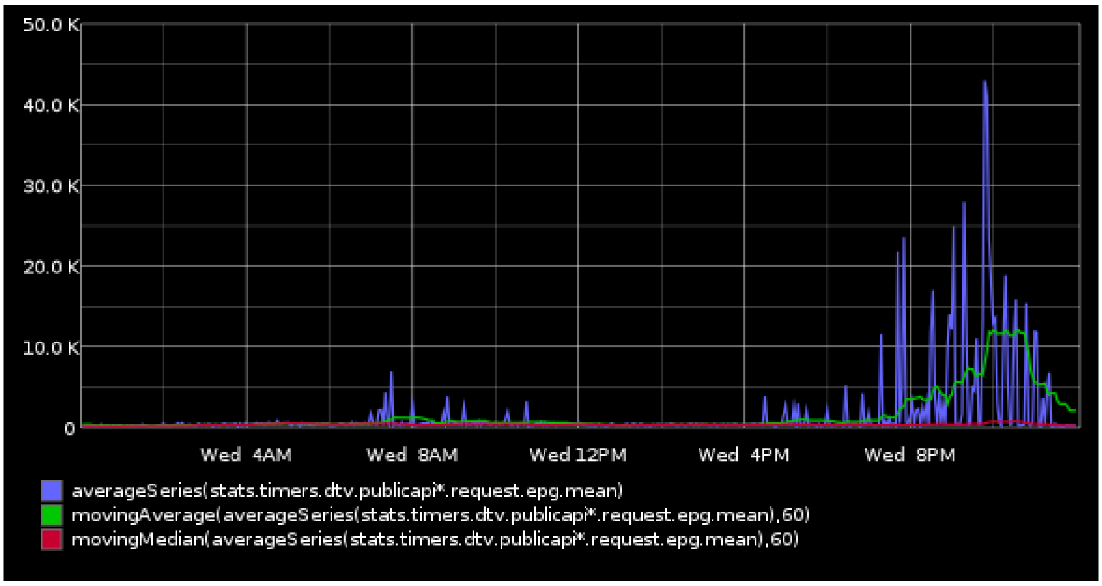


Figure 3.8: Moving average and moving median of a response time during incident

There are several problems with this approach, though. First of all, it requires rather expensive computation over all of the measurements in the window for each value, which becomes a significant overhead for larger windows. Secondly, it does not take into account any periodic variations or trends. Increasing the window size to enlarge the context will not help with that because the mean value will become insensitive to changes in the data. This, in turn, will result in constant alarms. Thirdly, this approach will lead to lots of false alarms on series where the usage during night is low and sporadic. Under these circumstances, insignificant a change in absolute numbers might mean a deviation of several hundred percent from the average.

To take the weekly periodic component into account, the author has often used additional *ad hoc* rules that also look at the value of the moving average from the same time window the week before. In order to raise an alarm, the moving averages of both the current and the last week have to differ enough from the current value.

Using the simple approach with moving window is good enough for some specific cases. Examples of such cases are response time measurements which usually do not have any periodic component and are rather stable under normal circumstances. Also, the response counters of web servers that are frequently updated and have slowly changing values, are good candidates for this approach. Sensitivity of the alarm trigger can be tuned by changing the window size, window function, comparison period and the percentage of difference.

3.7.5 Holt-Winters Exponential Smoothing

The author wanted to use something less *ad hoc* than the moving average with various extra rules, and decided to use the Holt-Winters exponential smoothing algorithm. This algorithm is often used for making predictions for the next value of the data serie based on the past values, and Graphite has an implementation of this algorithm built in.

In order to explain how the algorithm works we have to start with weighted moving average, which allows us to enlarge the averaging window, in order to take in more context without making the result too insensitive. Weighted moving average is defined in Equation 3.2. Where w_i is a weighting factor from a set of weighting factors that add up to 1.

$$s_t = \sum_{i=0}^{k} w_i x_{t-i} \tag{3.2}$$

It is useful to select weighting factors in a way that gives more weight to the latest measurements.

Exponential smoothing is another improvement on top of weighted moving average which is defined in Equation 3.3. Where a is a weighting factor that is chosen by the user and s_{t-1} is the estimate from the previous step.

$$s_t = aw_{t-1} + (1-a)s_{t-1} \tag{3.3}$$

The initial value s_0 can, for example, be set to the measurement w_0. Using exponential smoothing is computationally much cheaper than the moving average because at each step, one only needs the result from the previous step and the current measurement.

Holt-Winters exponential smoothing, in turn, is an improvement on top of the exponential smoothing algorithm. It is able to take into account seasonal variances and trends in the data by adding additional smoothing factors. A detailed explanation of this algorithm is outside of the scope of this work but a good step by step explanation can be found in (Office for National Statistics).

As an example of Holt-Winters smoothing in action, we will look at a sample incident shown in Figure 3.9.

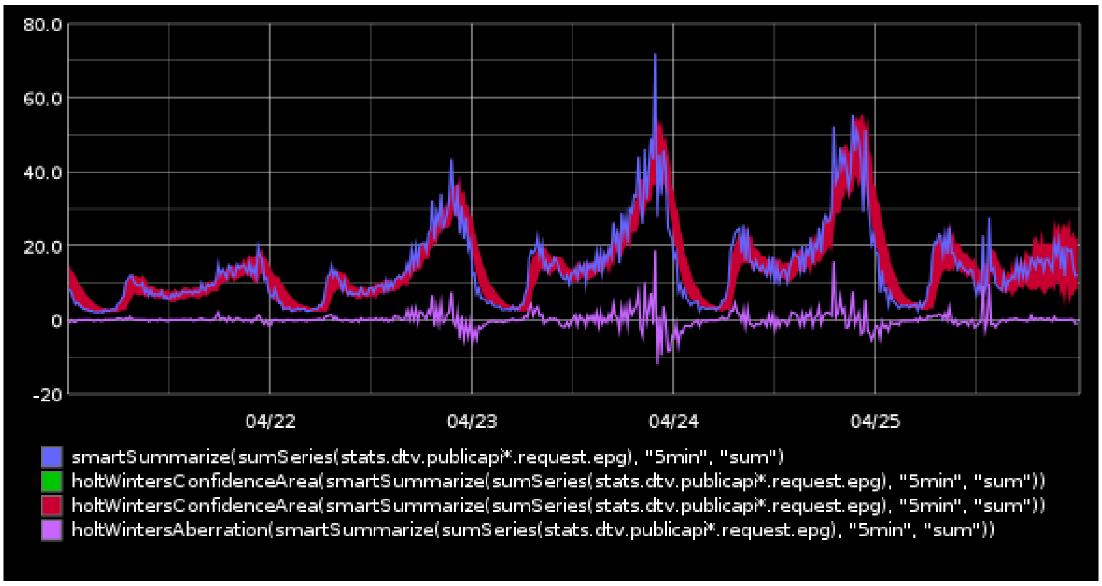


Figure 3.9: Holt-Winters confidence bands during an incident

The blue line on Figure 3.9 shows the number of requests per second against a specific URL in our IPTV REST API. The red area is a Holt-Winters prediction corridor where the value should be based on the historical data. The purple line shows the deviation between the predicted and real value. In this particular time frame, we can see a large disturbance as a new version of the software had unintended side effect of doing a lot more requests of certain type than before. We can see that the real value differs considerably from the prediction at various points during the incident.

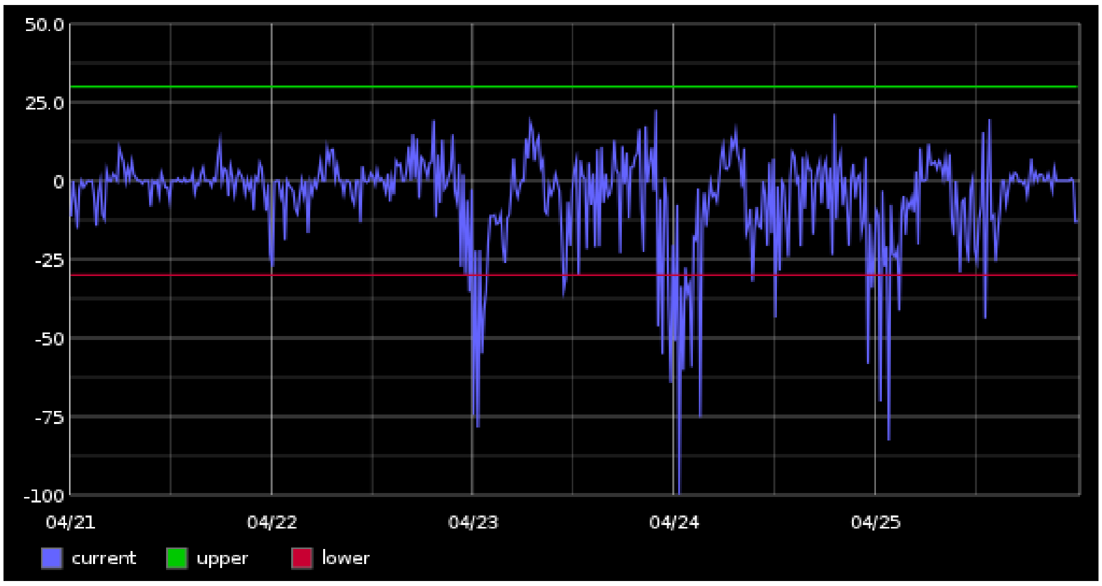


Figure 3.10: Monitoring graph with the alarm thresholds during an incident

Figure 3.10 shows the actual monitoring view of the same incident. The line *current* shows the difference of the current value from the one predicted by the Holt-Winters algorithm, as a percentage, and is defined in Listing 8.

```
alias(
  asPercent(
    holtWintersAberration(
      smartSummarize(sumSeries(stats.dtv.publicapi*.request.epg), "5min")
    , 2),
    smartSummarize(sumSeries(stats.dtv.publicapi*.request.epg), "5min")
  )
, "current")
```

Listing 8: Graphite query for getting Holt-Winters aberration in percentages

Each and every monitored graph is a bit different and it currently requires some thought and experimentation to define monitored graphs in a way that does not provide too many false positives.

Chapter 4

Results

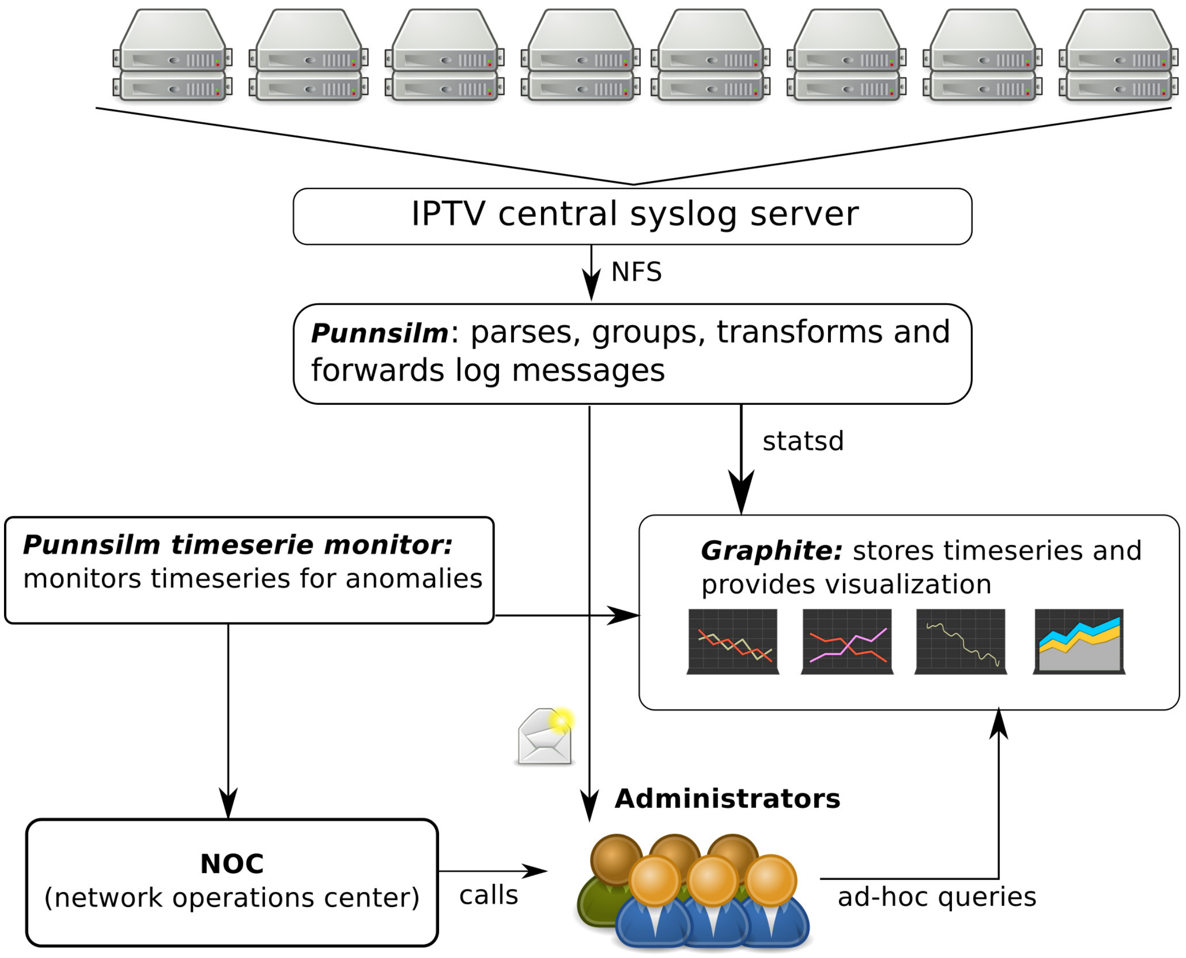
The author's initiative of extracting more value from the logs was a grassroots project, rather than something mandated from the top. With this in mind, the greatest success of the project has been the organizational acknowledgement that analyzing system logs is indeed really useful, and we should do more of it. This has been demonstrated by several internal awards that this project has won, and interest from other teams. Regardless of specific tools selected by different teams, the end result will certainly be a better quality of service for our end users. So far two other projects besides IPTV have started using Punnsilm and Graphite.

Punnsilm has been in use in our IPTV environment since spring 2013. It has helped us detect many issues that we otherwise would probably have noticed a lot later. While it is impossible to measure how long exactly it would have taken to notice these issues without Punnsilm, we have an example of a specific type of issue that took more than 70 hours to notice before Punnsilm was put into use. A similar problem was noticed and resolved in less than an hour afterwards.

Far more important than just saving time, is that a shorter incident discovery time leaves less time for customers to encounter that particular problem. This, in turn, should lead to a better user experience.

Both administrators and, to a lesser degree, developers have adopted the new tools into their daily workflow. IPTV administrators have dedicated a special screen in their room to live Punnsilm alarm logs and pay special attention to it after each change. We have had to roll back several software upgrades and configuration changes because of problems noticed that way. Figure 4.1 describes the deployment of the solution described in this thesis in the context of the IPTV project.

Better understanding of the importance of logs has also led to several efforts of making logs more structured and meaningful. For example, one of the projects that started to use Punnsilm, did a sub project for unifying logging methods over their code base, setting up central log server, adding RFC 5424 structured data fields to the messages and using reasonable values for the priority field. Such approach gives much better results and leads to simpler parsing possibilities.



IPTV central syslog server
NFS
Punnsilm: parses, groups, transforms and forwards log messages
statsd
Punnsilm timeserie monitor: monitors timeseries for anomalies
Graphite: stores timeseries and provides visualization
Administrators
NOC
(network operations center)
calls
ad-hoc queries

Figure 4.1: Overview of the Punnsilm deployment in the IPTV project

4.1 Performance

Currently, the most loaded Punnsilm installation is the one monitoring our IPTV solution. Anoter, a bit smaller installation updates last login timestamps in the SQL database for all all the IMAP and POP3 sessions that are served from our public e-mail services (hot.ee and suhtlus.ee).

Our IPTV installation parses around 20GB of logs per day that are sent from more than 40 different servers. The current configuration of Punnsilm contains 47 nodes that contain almost 200 regular expressions in total. The average flow of log messages is around 1500 lines per second. Over 4000 time series are extracted from these logs continuously and stored in Graphite. 1600 of these time series contain counters and 2500 contain timers.

Punnsilm runs on a dedicated virtual machine in the IPTV infrastructure that has 2 cores running at 2.4Ghz and has 2GB or RAM. The load average of that machine is shown in Figure 4.2 and a more detailed CPU usage breakdown is shown in Figure 4.3.

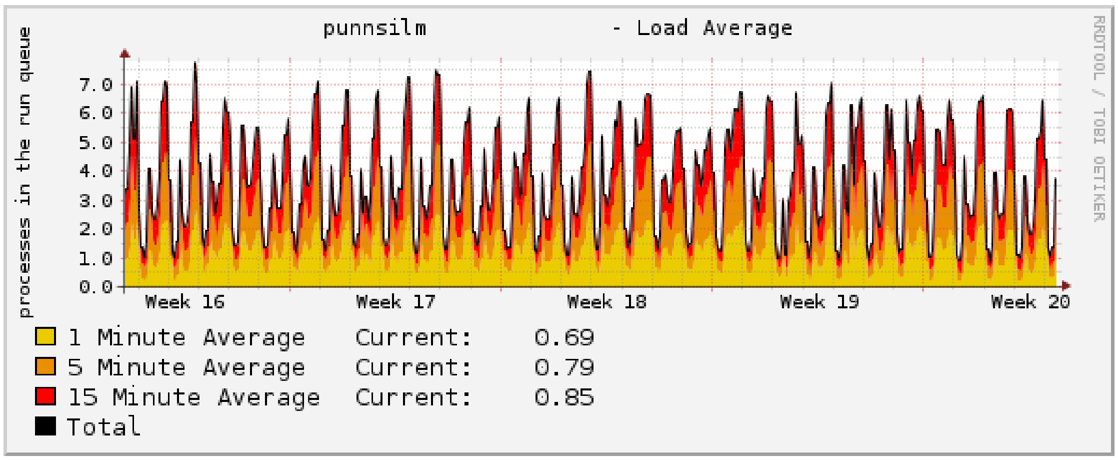
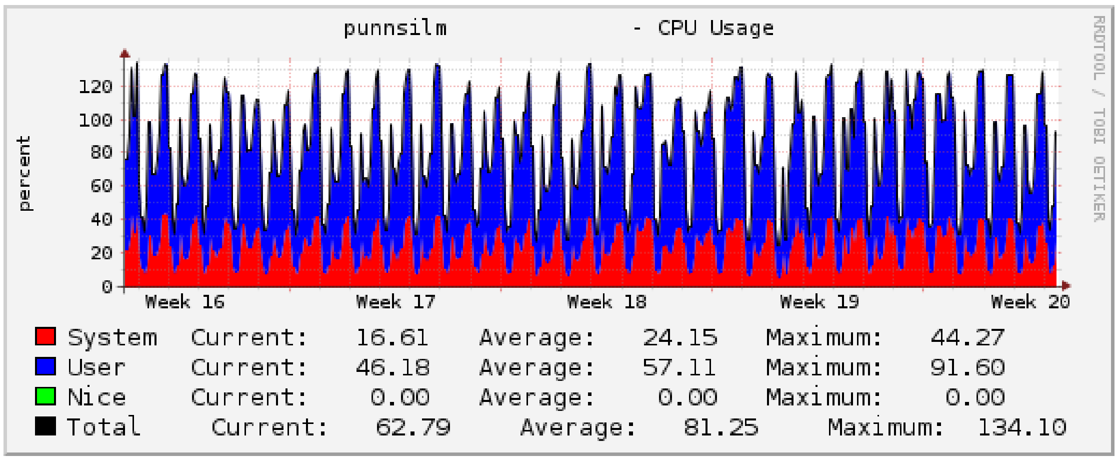


Figure 4.2: Monthly system load average of the server running Punnsilm



🟥 System	Current:	16.61	Average:	24.15	Maximum:	44.27
🟦 User	Current:	46.18	Average:	57.11	Maximum:	91.60
🟩 Nice	Current:	0.00	Average:	0.00	Maximum:	0.00
⬛ Total	Current:	62.79	Average:	81.25	Maximum:	134.10

Figure 4.3: Monthly CPU usage breakdown between userland and the kernel of the server running Punnsilm

In our mail systems, Punnsilm currently runs on the same servers that generate the logs. In total, it parses around 1000 messages per second in that installation and does around 20 SQL updates per second.

4.2 The Problems Discovered

Punnsilm has discovered more than 30 issues in our IPTV installation during the first 7 months of use. After this period of time, we stopped keeping track but in general, we discover at least one problem per week. Currently, the discovered issues have mainly been uncovered by the Artificial Ignorance approach and hardcoded notification rules. Because our IPTV network is closed to the outside world, various automatic exploitation tools that normally cause problems for this approach, have not been a problem for us. Time serie monitoring has also uncovered several interesting anomalies but is still a bit too noisy.

The problems discovered so far can be classified into the following categories:

1. resource starvation: This manifests itself in servers returning various unexpected error codes and response timings going up unexpectedly.

2. configuration issues: This includes cases such as

    (a) problems in the configuration of the proxy server that led to problems in corner cases;

    (b) problems in the configuration of content where files that were not present were referenced from the UI;

    (c) running into license limitations of the proprietary appliances;

3. content corruption: Several instances of video asset corruption were noticed, that led to discoveries of different deeper underlying problems with encoding parameters of specific channels, and in several cases, indicated developing hardware problems.

4. communication problems with other internal services: Several unexpected return codes and recurring availability issues were noticed and fixed that would otherwise probably have taken much longer to notice.

5. bugs in the code: Under some circumstances, broken URLs were constructed by the client software that had gone unnoticed for over a year. Several instances of requests to the development environment from the production environment were noticed.

Chapter 5

Summary

System logs are an often neglected source of information about the health of systems, that is not available from any other source. Problems can often be noticed quicker and the causes of known problems can be found faster, when this information is put into effective use.

The goal of this thesis was to find a solution for monitoring and visualizing system logs of various systems at Elion Ettevõtted AS. The introductory chapter discussed reasons why log analysis is hard, which is the main reason why it is not widely used. A selection of relevant tools was discussed next, and some of these were selected as the building blocks for our solution.

The main part of the thesis described the requirements for our log analysis system and the environment in which it has to operate. The author chose the components for our solution and discussed the reasoning behind the choices. In our solution a lot of information is extracted from the logs. The storing and visualizing of this information is handled by an open source tool called Graphite. We also use several other smaller Graphite related projects in our solution. The author decided to create a custom tool called Punnsilm which primarily deals with extracting information from the logs, detecting anomalies and forwarding information to other systems. The primary reason for creating our own software was that none of the open source tools that we discussed, were able to fulfill all of the requirements. Splitting the required functionality between different tools would have led to the duplication of similar configuration which is a violation of good practices. Punnsilm has been open sourced under the MIT license.

Anomalies are primarily detected from the logs using the Artificial Ignorance method. The idea of this method is that one should describe all the expected messages in the system, and everything that remains must be anomalous. In the process of matching the message against the ruleset, Punnsilm extracts a lot of structured information (request response times, counts of different request types etc), which is stored in Graphite as time series. This thesis describes how to use this information to find the root causes of the problems, and discusses how Punnsilm can be used to monitor these time series.

The resulting solution has been in production use for over a year, and has helped us find and debug many problems. Several deployments of Punnsilm at Elion in total, handle thousands of log messages per second from more than 50 different servers. Usually, at least one new problem is discovered per week thanks to Punnsilm, and the deployed visualization tools are in daily use by administrators.

In conclusion, the author's grassroots log analysis project has been a great success which has been confirmed by several internal awards and increasing interest from other teams.

Kokkuvõte

Süsteemilogid on tihtipeale teenimatult vähekasutatud infoallikaks süsteemi üldise tervise kohta – need sisaldavad infot, mida pole üheski muus allikas. Selle informatsiooni efektiivne kasutamine võimaldab kiiremini avastada süsteemides tekkinud probleeme ja leida väiksema vaevaga nende juurpõhjuseid.

Töö eesmärk oli leida lahendus süsteemilogide jälgimiseks ja visualiseeerimiseks Elion Ettevõtted Aktsiaseltsis. Sissejuhatavas osas käsitleti põhjuseid, miks logide analüüsimine on keerukas – keerukus on selle lähenemise vähese levimise peamiseks põhjuseks. Valdkonna ülevaate osas käsitleti erinevaid logihalduse seisukohalt olulisi avatud lähtekoodiga projekte, millest osa leidsid kasutust lahenduse ehituskividena. Töö põhiosa kirjeldas logihalduse lahendusele püstitatud nõudeid ja keskkonda, kus see lahendus toimima peab. Järgnevalt valis autor välja komponendid ja selgitas valikute tagamaid.

Logides olev informatsioon eraldakse erinevatesse aegridadesse ja nende talletamiseks ja visualiseerimiseks kasutatakse kirjeldatud lahenduses avatud lähtekoodiga tarkvara Graphite. Lisaks on lahenduses kasutusel ka mitmeid väiksemaid Graphite'iga seotud teisi komponente. Struktureeritud informatsiooni parsimiseks logidest, anomaaliate tuvastamiseks ja informatsiooni edastamiseks välistesse süsteemidesse otsustas autor luua oma tarkvara, mille nimeks sai Punnsilm. Peamine põhjus oma tarkvara loomiseks on, et ükski käsitletud olemasolevatest vahenditest ei suutnud täita kõiki soovitud funktsioone. Samas erinevate vahendite kasutamine nende funktsioonide täitmiseks oleks viinud sisult samade seadistuste kordamisele erinevate rakenduste jaoks. Punnsilm on tänaseks avalikult saadaval MIT litsentsi all.

Logidest tuvastatakse anomaaliaid eeskätt välistamismeetodil (*Artificial Ignorance*). Välistamismeetodi sisuks on süsteemis kõigi ootuspäraste sõnumite kirjeldamine ja reeglistikuga mitte sobivate sõnumite käsitlemine anomaaliatena. Reeglistikuga sobivuse uurimise käigus eraldatakse sõnumist ka üsna palju stuktureeritud informatsiooni (päringutele vastamise ajad, erinevat tüüpi päringute hulgad jne.), mis talletatakse aegridadena Graphiteis. Töö kirjeldab selle informatsiooni abil probleemide

juurpõhjuste otsimist ja selgitab, kuidas Punnsilma abil selliseid aegridu monitoorida.

Töö tulemusi käsitlevas osas kirjeldatakse erinevaid keskkondi, kus töös kirjeldatud lahendus on kasutusel. Põhiliseks selliseks keskkonnaks on Elioni IPTV projekt, kus Punnsilm analüüsib päevas umbes 20GB logi. Keskmiselt analüüsitakse selles projektis umbes 1500 sõnumit sekundis ja eraldatud info põhjal on Graphiteis loodud rohkem kui 4000 erinevate aegrida.

Projekti saab lugeda väga edukaks. Ühest küljest näitab seda avastatud probleemide suur arv ja see, et IPTV administraatorid kasutavad vahendit igapäevaselt. Teiseks on tekkinud projekti vastu huvi nii firma sees kui ka kontsernis laiemalt, mis on muuhulgas viinud ka mitmete firmasiseste auhindadeni.

Bibliography

S. Allen. Importance of understanding logs from an information security standpoint, 2001.

T. Atkins. Swatch readme. http://sourceforge.net/p/swatch/code/2/tree//trunk/README, 2006. [WWW] (2014-01-18).

D. Collier-Brown. Sherlock holmes on log files. http://datacenterworks.com/stories/antilog.html. [WWW] (2014-01-19).

C. Davis. Graphite. In A. Brown and G. Wilson, editors, *"The Architecture of Open Source Applications"*, pages 432+. lulu.com, June 2011. ISBN 1257638017. URL http://www.aosabook.org/en/.

R. Fekete. Log message classification with syslog-ng. http://lwn.net/Articles/369075/, 2010. [WWW] (2014-01-23).

R. Fekete. Correlating log messages with syslog-ng. http://lwn.net/Articles/424459/, 2011. [WWW] (2014-04-28).

R. Gerhards. The Syslog Protocol. RFC 5424 (Proposed Standard), Mar. 2009. URL http://www.ietf.org/rfc/rfc5424.txt.

S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *LISA*, 1993.

G. Jakobson and M. Weissman. Alarm correlation. *Network, IEEE*, 7(6):52–59, 1993.

J. Jenkins. Velocity culture. http://assets.en.oreilly.com/1/event/60/Velocity%20Culture%20Presentation.pdf, 2011. [WWW] (2014-01-05).

C. Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), Aug. 2001. URL http://www.ietf.org/rfc/rfc3164.txt. Obsoleted by RFC 5424.

I. Malpass. Measure anything, measure everything. http://codeascraft.com/2011/02/15/measure-anything-measure-everything/, 2011. [WWW] (2014-01-19).

Office for National Statistics. Annex b: The holt-winters forecasting method. http://www.ons.gov.uk/ons/guide-method/user-guidance/index-of-services/index-of-services-annex-b--the-holt-winters-forecasting-method.pdf. [WWW] (2014-05-11).

A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2007.

M. J. Ranum. artificial ignorance: how-to guide. http://www.ranum.com/security/computer_security/papers/ai/, 1997. [WWW] (2013-12-30).

M. J. Ranum. System logging and log analysis. http://www.ranum.com/security/computer_security/archives/logging-notes.pdf, 2005. [WWW] (2014-01-18).

K. Thompson. Logsurfer. http://www.crypt.gen.nz/logsurfer/. [WWW] (2014-01-19).

R. Vaarandi. *Tools and Techniques for Event Log Analysis*. Tallinn University of Technology, 2005.

R. Vaarandi. Mining event logs with slct and loghound. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 1071–1074. IEEE, 2008.

W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

T. Yarkoni. The homogenization of scientific computing, or why python is steadily eating other languages' lunch. http://www.talyarkoni.org/blog/2013/11/18/, 2013. [WWW] (2014-05-02).

Appendix A

Artificial Ignorance Example with Punnsilm

```
SSHD_TAG = "^sshd\[\d+\]: "

NODE_LIST = [
    {
        'name': 'syslog_source',
        'type': 'syslog_file_monitor',
        'outputs': ['filter',],
        'params': {
            'filename': 'auth.log',
        }
    },
    {
        # ignore known good sshd messages
        'name': 'filter',
        'type': 'rx_grouper',
        'params': {
            'groups': {
                'ignore': {
                    'rx_list': [
                        SSHD_TAG + "Accepted password for",
                        SSHD_TAG + "pam_unix\(sshd:session\): session opened for",
                        SSHD_TAG + "Received disconnect from ",
                        SSHD_TAG + "pam_unix\(sshd:session\): session closed for user",
                        SSHD_TAG + "Connection closed by",
                    ],
                    'outputs': [],
                },
                '_fallthrough': {
                    'outputs': ['console'],
                },
            },
        },
    },
    {
        'name': 'console',
        'type': 'console_output',
        'outputs': [],
    },
]
```

auth.log contents

```
Apr 13 11:11:23 hadara-laptop2 sshd[17471]: Accepted password for hadara from 127.0.0.1 port 39263 ssh2
Apr 13 11:11:23 hadara-laptop2 sshd[17471]: pam_unix(sshd:session): session opened for user hadara by (uid=0)
Apr 13 11:11:24 hadara-laptop2 sshd[17582]: Received disconnect from 127.0.0.1: 11: disconnected by user
Apr 13 11:11:24 hadara-laptop2 sshd[17471]: pam_unix(sshd:session): session closed for user hadara
Apr 13 11:11:35 hadara-laptop2 sshd[17683]: Invalid user joe from 127.0.0.1
Apr 13 11:11:35 hadara-laptop2 sshd[17683]: input_userauth_request: invalid user joe [preauth]
Apr 13 11:11:36 hadara-laptop2 sshd[17683]: pam_unix(sshd:auth): check pass; user unknown
Apr 13 11:11:36 hadara-laptop2 sshd[17683]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser=
rhost=localhost
Apr 13 11:11:38 hadara-laptop2 sshd[17683]: Failed password for invalid user joe from 127.0.0.1 port 39267 ssh2
Apr 13 11:11:39 hadara-laptop2 sshd[17683]: Connection closed by 127.0.0.1 [preauth]
```

Punnsilm output

```
$ punnsilm --config simple_ignorance.py --no-state
h:hadara-laptop2 ts:2014-04-13 11:11:35 content:sshd[17683]: Invalid user joe from 127.0.0.1
h:hadara-laptop2 ts:2014-04-13 11:11:35 content:sshd[17683]: input_userauth_request: invalid user joe [preauth]
h:hadara-laptop2 ts:2014-04-13 11:11:36 content:sshd[17683]: pam_unix(sshd:auth): check pass; user unknown
h:hadara-laptop2 ts:2014-04-13 11:11:36 content:sshd[17683]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
tty=ssh ruser=
rhost=localhost
h:hadara-laptop2 ts:2014-04-13 11:11:38 content:sshd[17683]: Failed password for invalid user joe from 127.0.0.1 port 39267 ssh2
```

Appendix B

Statsd Example with Punnsilm

This is an example of configuration for Punnsilm that builds regular expression out of smaller regular expressions in different variables. Three time series are extracted from the matching loglines and sent to statsd.

```
# obviously misses some legal IPs and matches some
# non-legal but is good enough for our specific case
IPV4 = """[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+"""
NGINX_TIMESTAMP = '(?P<nginx_timestamp>[0-9]+/[A-Z][a-z]{2}/[0-9]+:[0-9]+:[0-9]+:[0-9]+ \+[0-9]+)'
STATUS_CODE = "(?P<_http_code_value>[0-9]+)"
BYTES_RETURNED = "(?P<bytes_returned>[0-9]+)"
TOTAL_TIME = "(?P<_ref_controller_value_time>[0-9]\.[0-9]+)"
BACKEND_TIME = "(?P<_backend_time>[0-9]\.[0-9]+)"
REFERER = "(?P<referer>[^ ]+)"
USER_AGENT = '(?P<user_agent>[^"]+)'

HTTP_METHOD = '(?P<http_method>[A-Z]+)'
PROTOCOL_VERSION = '(?P<protocol>[A-Z]+)/(?P<http_version>1.[0-9]+)'
REQ = HTTP_METHOD+""" (?P<http_uri>/api/index/et/(?P<_controller_value>[a-z]+)[^ ]?) """+PROTOCOL_VERSION

NGINX_RX = 'nginx: '+IPV4+' - '+IPV4+' - - '+NGINX_TIMESTAMP+' "'+REQ+'" '+STATUS_CODE+' '+BYTES_RETURNED+\
    ' '+TOTAL_TIME+' '+BACKEND_TIME+' "'+REFERER+'" "'+USER_AGENT+'" .'

NODE_LIST = [
    {
        'name': 'syslog_source',
        'type': 'syslog_file_monitor',
        'outputs': [
            'nginx_parser',
        ],
        'params': {
            'filename': 'testlog1.log',
        }
    },
    {
        'name': 'nginx_parser',
        'type': 'rx_grouper',
        'params': {
            'groups': {
                'group1': {
                    'rx_list': [
                        NGINX_RX,
                    ],
                    'outputs': ['statsd',],
                },
            },
        },
    },
    {
        'name': 'statsd',
        'type': 'statsd_output',
        'params': {
            'key_prefix': 'test',
        },
        'outputs': [],
    },
]
```