Building A Better Network Monitoring System

A report
submitted in fulfillment
of the requirements for the degree
of
Bachelor of Computing and Mathematical Sciences with Honours
at
The University of Waikato

by

Brad Cowie

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Department of Computer Science
Hamilton, New Zealand
October 23, 2012

Abstract

Network Monitoring Systems are essential in running the complex computer networks of today. They ensure all faults on the network are known and assist the network operator in fixing these faults. There are problems with the typical NMSs used for monitoring networks, both with the difficulty of configuration and the age of these systems. The popular NMSs in use today are either themselves old or extended from old systems designed a decade ago for monitoring much simpler and smaller networks.

This project aims to plan and implement a system designed from the start to monitor a modern network and includes all tools in a single extendable system. By using today's technologies and making good design decisions, this project aims to build a unified and fresh NMS that is easy to configure, maintain and use, streamlining the workflow of a network operator.

Acknowledgements

I would like to acknowledge and thank my supervisor Richard Nelson for supervising this project. Big thanks to the staff at the WAND research group for their support throughout the project, especially Shane Alcock and Brendon Jones for proofing parts of this report. Also to the past and present students of WAND, thanks for the support and tea breaks over the last number of years.

Thanks also to Jamie Curtis for the suggestion of the project and motivation for carrying out this work.

Contents

List of Figures

List of Acronyms

IP Internet Protocol

NMS Network Monitoring System

ISP Internet Service Provider

SOHO Small Office Home Office

SNMP Simple Network Management Protocol

OID Object Identifier

UPS Uninterruptible Power Supply

RRD Round-Robin Database

RRA Round-Robin Archive

MRTG Multi Router Traffic Grapher

RTT Round Trip Time

I/O Input/Output

DNMS Dynamic Network Monitoring System

API Application Programming Interface

REST REpresentational State Transfer

SMS Short Message Service

IM Instant Message

Nmap Network Mapper

Chapter 1

Introduction

A Network Monitoring System (NMS) is an essential aspect of running a computer network of a significant size. NMSs are used to identify faults before they happen and reduce the impact they may have on users of the network. Networks range in size from small office networks to large Internet Service Provider (ISP) networks, yet a NMS needs to be able to adapt and be scalable to a network of any size, while still providing the same level of insight without overwhelming the user with unnecessary information.

A NMS adds insight by employing a number of different monitoring techniques, including:

- service polling
- graphing
- event system
- notification system

These different systems interact to provide insight about the network being monitored. Insight is provided in two ways. First, instant feedback in the form of event notifications is provided for immediate problems. Second, historic data is regularly collected and archived for later analysis. Archived data will show up trends and can be used to help identify the root cause of an event found by the NMS.

A NMS provides both technical and business insight into the network. A NMS helps day to day by ensuring technical problems are addressed on the network. At the same time, a NMS will show where to spend money upgrading

and improving the network by highlighting the parts of the network that are reaching full capacity and need upgrades. Also, the NMS will reveal unreliable segments of the network that provide degraded service.

The popular NMSs used to monitor networks currently have a number of limitations imposed by legacy tools and libraries that are used in the construction of the systems. These limitations are due to constraints that were required in the past but are no longer applicable. One such limitation is the inability to store large amounts of data that has been collected by the NMS accurately. This is because of assumptions made about the data by the database format and due to the fixed size of the database. It is also common that a NMS will only implement a subset of the techniques used to monitor a network. This makes it necessary to deploy multiple NMSs to provide full testing coverage for a whole network.

This project explores newer technologies and techniques with the goal of producing a refreshed NMS that is not restricted by these same limitations. The new techniques investigated aim to simplify and streamline the installation and maintenance required to run a NMS. In this project we will examine the latest generation components that can be used to build a NMS. With these goals we investigate writing refreshed versions of:

- data collector
- data store
- data interface
- web interface
- graphs

DNMS is the system built during the project to implement some of the new techniques discussed. DNMS is a modular system implemented in a way that can be extended to include all techniques used to monitor a network, including the new techniques investigated. This project exists to unify the network monitoring ecosystem and allow one system to be used instead of the current situation where multiple NMSs are used.

This report explains the background and motivations behind a NMS in Chapter 2, which identifies the understanding and reasoning for further chapters. Chapter 3 investigates the NMSs used currently, both popular systems and

newer systems that have similar goals to this project. This chapter also dissects the tools and libraries used by these systems. Limitations and issues in these existing systems are discussed in Chapter 4, putting further emphasis on the original problem. In Chapter 5 we investigate solutions to the problems raised in Chapter 4 using new technologies and suggest further techniques that could be used to simplify network monitoring. Using the outcomes from this investigation we look at the implementation of DNMS using these methods. Conclusions and further work to extend this system and enable DNMS to become a feature complete NMS are discussed in Chapter 7.

Chapter 2

Background

2.1 Networks

Computer networks are a series of devices that are interconnected and able to communicate with each other. Typical computer networks are comprised of computers connected together by switches and routers. Switches provide the forwarding capability that allow logically neighbouring devices to communicate. Routers add the routing capability that provides the network with structure and allows communication between sub-networks. The type of network we are most interested in for this project are Internet Protocol (IP) [12] networks.

Computer networks vary significantly in size and importance. The larger and more important networks can cost the organisations that are running them large sums of money for every minute that they are unavailable or malfunctioning. Networks that fall into this category are ISP networks and high speed financial trading networks. ISP networks can have devices and customers distributed across an entire country and outages on core network components can cause a large number of customers to lose access to the services provided by that ISP. On a high speed financial trading network, such as a stock exchange, every second of uptime matters and minutes of downtime can be very expensive. On the other end of the spectrum, Small Office Home Office (SOHO) networks are considerably smaller than the previous examples, possibly only a few devices in size, however these networks can be equally important to the owners and users even though the direct cost of an outage is much less than an outage on an ISP network.

2.2 Network Monitoring Systems

When a network grows large, it becomes infeasible for one person to maintain a mental model of the entire network. This makes the network an unknown entity where faults could be happening at any time and not be detected by the network operator. A NMS is a software package used to solve this problem and diagnose faults on the network. A NMS achieves this by storing an internal model of what the network is supposed to be and uses this model to evaluate the current state of the network. In doing so a NMS can provide insight into the otherwise unknown entity. It will also provide performance data on how well the network is utilised and answer questions regarding the network economics, i.e is the network cost effective and meeting demand?

As with software engineering testing best practices, test coverage is also critical when monitoring a network. Having high testing coverage is going to reduce the amount of the network that is unknown and ensure that network faults are noticed. The test frequency should be high enough that information can be obtained in a timely fashion to identify and fix any problems that are observed. However it is equally important that the load generated by the NMS does not impact upon the device or service in a way that would affect the network users.

A number of different techniques can be used by a NMS to monitor a network. The rest of this section will focus on the features required in an ideal NMS and those that should be provided to produce what we will call a general purpose NMS.

2.2.1 Configuration System

To encourage a NMS user to have high testing coverage on their network it is important to have a well-designed configuration system. Networks change frequently making the network model held by the NMS outdated, resulting in "black holes" that are not tested or monitored. If it is easy to initialise and maintain the NMS configuration, the model of the network should be more accurate. An accurate model will ensure that the NMS will be able to monitor the network better.

2.2.2 Service Polling

Service polling is a type of test where the NMS checks regularly whether a device or a service is available and working within normal parameters. This allows the NMS to answer important questions such as whether or not the server that is hosting our website is reachable over the network. More specific polling tests allow the NMS to collect additional information about the current network state. An example of such a test would not only ensure that our web server is operational but also verify that the web server software on that machine is running correctly, is responsive, and is serving the correct content without any errors.

2.2.3 Graphing

Drawing time-series graphs of performance data can be useful for identifying trends and anomalies. Such graphs are frequently used for identifying changes given a large quantity of data values. However, graphs often need to be tailored to suit the network being monitored and some data is more useful to graph than others. One example of time-series data that is useful to graph is CPU usage over time, as this will indicate whether or not our devices are being overworked and how close they are to operating at full capacity. Graphing bandwidth usage on network interfaces will show how close the network is to running at full capacity as well. This information can be used by the network operator to plan network upgrades and identify likely bottlenecks.

2.2.4 Notifications & Events

When there is a change on the network, good or bad, someone should be notified of the change. In a NMS this is done with an event system. Event systems can be very simple systems that check if some value is within a given threshold or has a certain value. For example, an event may trigger if a host becomes unreachable. Event systems can also be more complex and watch for undesirable trends or use anomaly detection to identify events that could impact the network. Often, the notification system alone does not provide the full picture and some expertise may be required to determine what has caused an event to trigger. This is because there are metrics, such as load average, which are a combination of many different measurements including Input/Output (I/O) activity, CPU usage and memory usage. When a load average reaches a high

value and an event is generated it may not be immediately clear what caused the event and how to fix it. A NMS will allow the user to find the root cause by providing links to historical time-series performance data that will indicate where the problem lies.

A good notification system will only send out notifications that are actionable, i.e every notification that is sent out by the notification system is a problem that an operator can resolve. A notification that cannot be acted upon is a waste of time and energy for the person who receives it and makes it more likely they might ignore a notification in the future. Ideally the notification system will also try to avoid simply generating one notification per event. For example, a smart notification system will prevent multiple notifications being sent for flapping events where the same event switches between a good and bad state repeatedly over a short period of time.

2.2.5 Dashboard

A dashboard is an interface that provides a visual display of information using a single screen such that all this information can be monitored at a glance [7]. A dashboard is useful for a NMS because it allows the whole network to be monitored from a large monitor in view of the network operator and does not require actively searching through many pages of a NMS to manually identify problems. A typical dashboard will include graphs that should be checked regularly for that network, for example graphs of performance on core network links or the number of active users of core systems. It will also include summary statistics such as the current number of faults or a generalised network health measure that consists of a wide range of factors that describe the current performance of the network as a single entity.

The dashboard should streamline the process of finding faults by ensuring that it is easy to discover the root cause of an event and by pointing the user in the direction of where they should start any further investigations.

Chapter 3

Prior Work

NMSs have existed in one form or another since the first networks were built. Initially they were haphazard collections of scripts but eventually developed into more complete and professional systems that were released as open source software. As a result, the networking community could benefit from the work of others and contribute their own scripts as well. One of the first such systems was Nagios [16] in the mid nineties [9]. Since Nagios, many further systems have been written in an attempt to produce the ideal NMS, creating a plethora of NMSs. Most of them have been created by System Administrators who prefer to modify and extend working systems rather than engineering software from scratch. This has resulted in a very disparate ecosystem in which many separate tools are required to monitor a network. This is shown in Figure 3.1 which depicts the architecture diagram for MetaNAV [32], a NMS built from many different systems that attempts to combine the data into a single database and provide analysis from this single source.

3.1 Common Tools and Libraries

There are a number of common tools and libraries that are used to build NMSs. The items covered in this section are industry standard at present and are used by all of the popular systems covered in the next section. In this section we will look at how these tools and libraries work and the part they play in building a NMS.

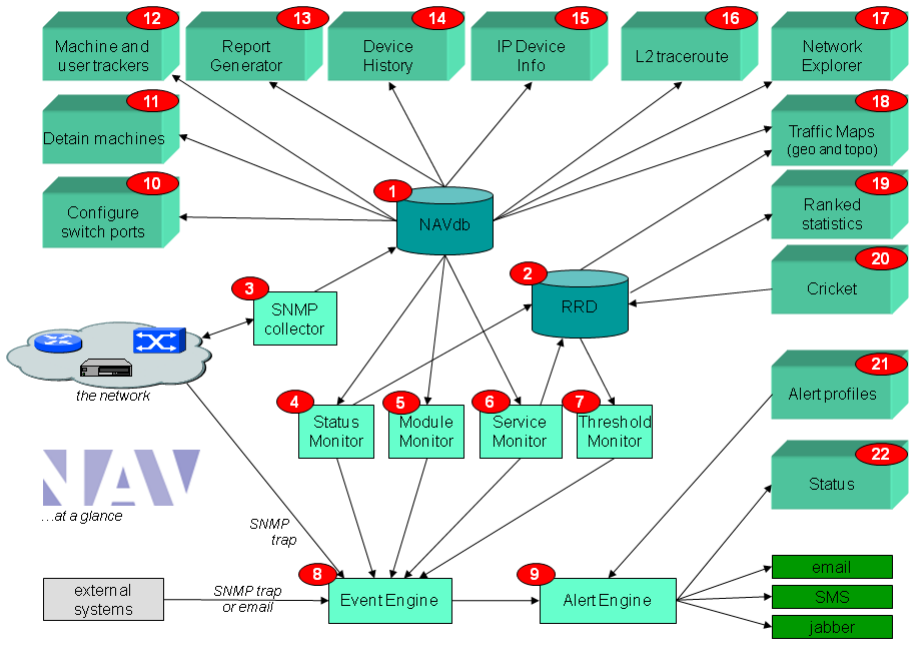

Figure 3.1: Architecture diagram of MetaNAV

3.1.1 Data Collection

Almost all NMSs utilise Simple Network Management Protocol (SNMP) [4] for data collection. The reason for this is the ubiquitous nature of SNMP. Nearly every device on a network is guaranteed to be able to be monitored using SNMP. Devices implement SNMP because, as the name suggests, it is a very simple protocol with minimal overhead that can be implemented in resource constrained environments. SNMP is primarily a query based protocol where a client requests specific management or performance data from the servers in a query.

Each query is centred around an Object Identifier (OID) that is part of a hierarchical numbering scheme that collects related data together. The management protocol component of SNMP has the ability to not only read from OIDs but also to write values back to them to control the remote device in some way. SNMP also has limited support for pushing information to a remote device in case an event occurs that requires an action before the next poll interval occurs. For example, if an Uninterruptible Power Supply (UPS) loses power, it is important to know instantly rather than wait potentially minutes for this information to be polled.

3.1.2 Data Archival

The current state of network data archival is similar to that of data collection. Most NMSs tend to store their time-series data in Round-Robin Databases (RRDs) [19]. RRDs have been in widespread use since 1999 [22] and were designed as a faster and more configurable replacement for Multi Router Traffic Grapher (MRTG) [18] which was the standard at the time. The major advantage of storing time-series data in a RRD is that they are small and fixed size and are therefore ideal for space constrained monitoring systems. This was a sensible approach when RRD was designed as disk storage was both small and expensive.

Measurements are stored in a fixed size database by only storing a fixed number of entries in a Round-Robin Archive (RRA). A single RRD is typically configured to have multiple RRAs and data values can be aggregated from minute intervals all the way out to summaries for years. All configuration for a RRD, such as the time interval that data will be arriving at, is permanently set inside the file and cannot be changed without creating a new RRD. This is problematic because as networks grow and expand the network monitoring system needs to do the same; if a network link is increased in capacity we need our database to represent this.

3.1.3 Graphing

As a result of RRD being the prominent data archival solution for NMSs, most systems use the graphs generated by RRDtool [20]. RRDtool is the official and reference implementation of RRD and can be used to create new databases, configure the metadata inside a database, update and store new values in a given database, fetch data values from inside a given database and print metadata for a given database. As well as supporting the required functions to interact with RRDs, RRDtool also includes the ability to draw graphs of time-series data stored in a RRD. The graphs can be generated using several different graphic formats: Portable Network Graphics (PNG), Scalable Vector Graphics (SVG), Encapsulated PostScript (EPS) and Portable Document Format (PDF). RRDtool only generates a static graph; it does not supply an interface for interacting with these graphs. Exploratory features such as panning and zooming are left up to the developer of a system to implement but are often missing.

A simple RRDtool graph of a single data point is shown in Figure 3.2.

3.2 Network Monitoring Systems

In this section, we discuss some of the more popular NMSs that are in use today. Popular systems were picked from the Debian Popularity Contest [27], which tracks the number of installs of Debian packages. The examined NMSs were evaluated at the start of this project to determine which projects were similar to my project and to examine the current state of the NMS field. Two newer NMSs, Graphite and Reconnoiter, were also examined. These NMSs are rewrites from scratch of the traditional NMS stack and try to do something different than the status quo for network monitoring. It is important that a new NMS would be able to perform similar tasks to those NMSs examined here to encourage adoption of a new system.

3.2.1 Cacti

Cacti [30] is a NMS designed for drawing time-series graphs of performance data on a monitored network. Cacti typically draws a different graph for each monitored data source. A graph can be drawn from multiple data sources, but requires a suitable template created using the cacti format. The configuration system is entirely web based and there is no provided method for performing bulk configuration. The additional effort required to update the network model in Cacti often discourages the user from monitoring everything. Cacti also does not include an event detection system or a notification system and therefore is usually used to supplement another NMS by providing historical graphs. These provide more visibility and insight as to why an event may have triggered in another monitoring system.

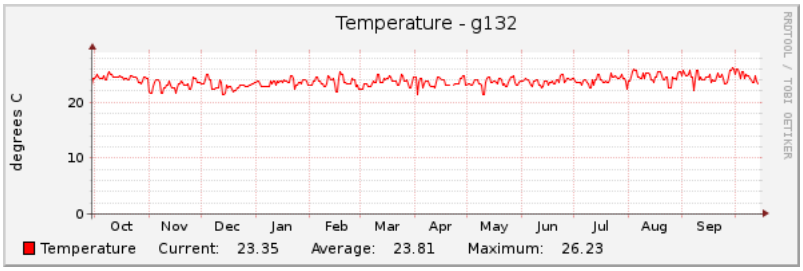

Figure 3.2: RRDtool graph of the ambient temperature in my lab for the year of 2012

Data to be graphed in Cacti is collected using SNMP at a specified rate. This defaults to 5 minutes but with some effort can be reduced to a faster rate. This data is then archived by storing each data source in separate RRD files.

3.2.2 Icinga

Icinga [31] is a NMS designed specifically for service polling, notifications and report generation. Icinga is a fork of Nagios that was released in 2009 [6] and uses a large portion of Nagios code still in its core. Icinga service polling is modular: each service check is handled by a separate check script or process which is forked by the Icinga monitoring system. The check process will exit with an exit code that matches the severity of a problem (ok, warning or critical). Performance data from the test can be reported to Standard Output (stdout). Since Nagios has been an industry standard for the past ten years and is easy to script and modify, there are many different community maintained extensions that can extend Icinga and Nagios to support extra features. For example, there is an extension for graphing all of the performance data that is collected from check scripts, which is a feature not available in the core code.

Data collection in Icinga is handled by the check script and any protocol could be used to collect measurement data. The nagios-plugins [17] package, which is a bundle of commonly used check scripts for Icinga and Nagios, exclusively uses SNMP to collect data.

Data in Icinga is stored in plain-text log files and parsed every time the system needs to access historical data. There are extensions to Nagios and Icinga that can store this data in a Structured Query Language (SQL) database instead of plain-text files, producing faster query times.

3.2.3 Smokeping

Smokeping [21] is not a conventional NMS, especially compared with Cacti or Icinga. It is a tool that provides more specific monitoring that is critical inside ISPs to enable Network Administrators to identify faults and performance problems on network links. Smokeping monitors latency on a network by sending periodic Internet Control Message Protocol (ICMP) echo request packets to a machine at the other end of a network link and waiting for that machine to send back an ICMP echo reply packet. The Round Trip Time (RTT), which is the time difference between the request and corresponding reply, is recorded

following each test, as well as any instances where there is no successful response. This type of test is invaluable for monitoring networks as it reveals many different types of faults with just two checks. An increase in RTT, for instance, can indicate a change in network routing that is causing packets to travel a longer path. Data collected by Smokeping is stored in RRD files, with each monitored host having a separate RRD.

3.2.4 Graphite

Graphite [5] is a NMS written in Python with the aim of being more scalable and offering more real-time graphing capabilities than existing NMSs. Graphite includes an extendable data collection server that supports using user-developed plugins to supply data. There are no built-in plugins supplied for data collection; rather, it is intended that the user implements a data collector to suit their particular use case. Data is stored in a specialised database that is designed to be similar to the RRD format but without the limitations. For example, data can be provided at irregular intervals and still be stored. However one limitation remains in Graphite, it is restricted to only storing and working with numeric time-series data. Graphite configuration is provided by a minimal series of plain-text files. The web interface supplied by Graphite is capable of drawing custom graphs of the data that has been collected. However, Graphite does not provide a dashboard or event detection system, which are both essential for a general purpose NMS.

3.2.5 Reconnoiter

Reconnoiter's [23] stated goal is to improve the NMS field by using fault detection and trending together to provide more information and insight than current systems. Reconnoiter is aimed at operators running devices in multiple datacentres in geographically different location. With this use case in mind, Reconnoiter aims to be more efficient other alternative NMSs by having a decentralised data collection system. The collectors are moved close to where the data is, in this case a single collector runs at each remote datacentre, and results are communicated back to a central statistics collection server that stores all of the data long term for analysis and graphing. The data is stored in a Relational Database Management System (RDBMS) as opposed to the RRD format.

Reconnoiter encourages good monitoring practices by having a robust configuration system. Reconnoiter is designed to have centralised configuration management and decentralised configuration manipulation [24]. This approach to configuration makes Reconnoiter flexible enough to be deployed at remote datacentres and still be easily configured from a central system.

Check scripts can be written in C or Lua and run on the remote collection servers. The metadata for a check is defined in Extensible Markup Language (XML), including arguments to be sent to the script as well as how to execute it. The remote data collection server then periodically executes the defined check scripts and passes the result back to the centralised data store.

Chapter 4

Issues With Existing Solutions

In the previous chapter we examined existing NMSs that are used to monitor networks. We also looked at the tools and libraries that they are built from and how these work. In this chapter we will identify common problems in these systems that make them unsuited to being used as general purpose NMSs. These issues are the reason that it is common to deploy multiple NMSs, each with their own strengths and features, to provide full monitoring coverage for a network.

4.1 Data Collection

SNMP, as mentioned in the previous chapter, is the industry standard for data collection in a NMS. SNMP is commonly configured in a centralised architecture, where a single SNMP collector collects data from every device on the network. In a paper studying the behaviour of SNMP collectors [26], issues are raised with the performance of the centralised design of SNMP when used for large-scale network monitoring. A better design suggested in the paper is a decentralised approach. With this approach, remote collector agents are used to monitor a subset of the network each. These collector agents periodically export data to a central system for storage. This approach has the benefit of working well when collecting data over the internet. Devices on a network are often held in different geographical areas for redundancy and moving the collector agent logically closer to where the devices are is beneficial. This is because a single expensive connection to a central storage server over the internet is better than one expensive connection per monitored device.

Another issue is that SNMP is exclusively used in a polling configuration.

SNMP can be used to push data with a SNMP trap, which is built into the protocol for pushing data immediately to a collector. However, due to the difficulty of configuring traps correctly, the feature mostly goes unused. This means that we rely on the poll interval of a SNMP collector being fast enough to detect important changes. However, monitoring a large network with a single SNMP collector will prevent a fast poll rate because of load and bandwidth constraints on the collector machine.

NMSs will bundle a SNMP poller that they will use to collect data. This is either unchangeable or difficult to change. If we are deploying multiple NMSs to provide the full suite of monitoring tools this means we have to run multiple SNMP pollers as well. Some SNMP implementations do not perform caching of values so polling values on a device multiple times can be expensive to that device. This goes against the requirement that our monitoring should not greatly impact the services of the network.

4.2 Data Store

The NMSs looked at in the previous chapter mostly used the RRD format to store time-series data. RRD is good at being a storage constrained solution because it is a fixed size and never grows. It is therefore easy to predict how much storage is required. However, storage is now very cheap compared with when RRDs were first proposed so we can afford to store more historical data. An important property identified for the NMS storage system is that data is archived for later analysis. RRD does not meet this requirement because it only stores a fixed number of data points in a round-robin fashion, keeping the most recent n entries.

Other data integrity issues exist with the RRD format as well. All inserts into the database must happen sequentially, meaning that there is no way to manually back fill data. Back filling is useful for importing historical data into the database, which could be useful for identifying long term trends. RRD also expects all data to be updated at regular intervals, such as every 5 minutes. If a new data point arrives a lot later than 5 minutes then the previous data point may be dropped. If it arrives some number of seconds after 5 minutes RRDtool will store an interpolated value for what the value would have been on the 5 minute interval based on past data points, instead of the raw data point that was received. This means that RRD cannot guarantee accurate and

consistent data in the event that data is collected at irregular intervals, which can be caused by delays on the network or in data collection. When we are using the data stored from our NMS to make important decisions on how to grow our network or determining how much to bill users on the network, we want accurate and consistent data.

4.3 Dashboard

From the systems outlined in the previous chapter, most of them have useful dashboards. This is because the dashboard is the interface that most users of the system deal with day to day. A NMS will not be used to monitor a network by a network operator if it has a bad interface.

The main issue with dashboards is that networks must be monitored by multiple NMSs to have full testing coverage over the entire network. This means that multiple dashboards need to be used to track the status of the network. A user will be glancing at multiple dashboards to scan for faults which can be difficult and distracting. A solution that can unify this all to exist on one dashboard would better satisfy the requirements outlined earlier.

4.4 Configuration

The most common configuration system used by the NMSs in the previous chapter are plain-text configuration files. These configuration files define the network model and how the NMS should be monitoring the network.

The issue with using plain-text files is that the amount of configuration required can be quite large. For example, a small Icinga instance that monitors 34 devices that was investigated during this project uses 3724 lines of configuration to define the network model. Configuration of this magnitude is very prone to human errors which can go unnoticed for extended periods of time. This negatively impacts on how well the network is being monitored. Another system, Cacti, does not make plain-text configuration accessible. Its configuration is locked in a custom RDBMS schema that is only accessible from a web interface. The web interface does not provide bulk configuration options, which adds to the effort required to configure a whole network. These problems stop us from meeting the requirement of encouraging users to have high

testing coverage by accurately setting up and maintaining the network model in the NMS.

Running multiple NMSs exacerbates the problem as the user must now maintain a number of different configuration systems. These configuration systems are incompatible. As a result, one change to the network model will mean updating this in multiple systems, further increasing the likelihood of mistakes.

Chapter 5

Investigation

It is apparent from previous chapters that there are a number of issues with the most popular NMSs that are used today to monitor networks. We also identified two newer NMSs that appear to be heading in the right direction to fix some of the problems identified. However, they are targeted to specialised monitoring and are missing important features such as notification systems and decent dashboards. Also, limitations such as only storing numeric time-series data make them unsuitable as a general purpose NMS because it is useful to track discrete string values.An example is the state of a Redundant Array of Independent Disks (RAID) array which could exist in a number of states such as healthy, degraded, failed or rebuilding.

In this chapter we will investigate new techniques and updates to current techniques that could be used to build a new NMS. These updated techniques will aim to fix problems and limitations raised in the previous chapter. This chapter will also introduce new methods that can be implemented in an NMS alongside current methods to improve the configuration and maintenance of the NMS configuration.

5.1 Data Collection

This section will define a new set of requirements for a new data collector to replace SNMP. By replacing SNMP we seek to remove the limitations that it adds to the data collection section of a NMS.

Most devices being monitoring by a NMS are powerful servers or switches. The requirement to have a very simple protocol that can work on low hardware re-

quirements is therefore unnecessary. Instead, we want an efficient protocol built for transmitting large amounts of data points regularly without a significant load impact on our collection server. A data collector with support for pushing data as a first class citizen allows important data to turn up instantly rather than on the next poll interval. A collector with a fast poll interval, to the point of being able to collect data at real-time without much overhead, will add another technique for a network operator to use to inspect active faults in greater depth. Using a decentralised system eliminates a single point of failure for our collector and improves scalability. Similar projects such as the two new NMSs discussed earlier are utilising decentralised data collection systems and have replaced SNMP with their own collector.

5.2 Data Store

This section will cover a new set of requirements for a new data store for use in a NMS to replace the widely used RRD format. By replacing RRD we seek to remove the limitations it adds to the storage of data in a NMS.

Hard drive prices are much cheaper compared with when RRD was originally designed. It is now common for dedicated monitoring servers to be purchased for monitoring large networks. These servers have abundant storage capabilities because the cost of storing historical data at the same resolution of new data is far more valuable than the cost of hard drives. The requirement to use a fixed-size data store that keeps n most recent data values can be relaxed. The data store for the new NMS should store all measurements. It should also be capable of recording multiple data types (strings, integers, floating point values).

RDBMSs have matured greatly since RRD was first popularised and since then have become a reliable and widely supported standard for storing large quantities of data. Using a RDBMS also enables the stored data to be interacted with using SQL. SQL is a powerful language for interacting with stored data and allows rich queries to be executed to analyse stored data. There are many other benefits of using a RDBMS, including the ability to split database storage. Recent data can be stored in memory for fast access while archived data, which does not require fast access, can be stored on the hard disk.

5.3 Automatic Configuration Discovery

One of our stated goals was to ensure the configuration is easy to initialise and maintain. The best method of achieving this goal is to limit the amount of configuration required by the NMS by performing automatic configuration where possible. In this section we will look at three different methods of automatically generating different parts of the configuration for a NMS.

Automatic configuration does not solve all configuration issues however. The automatic configuration must be maintained through some mechanism as well. This maintenance is too expensive to run with every interaction with the NMS so an update interval must be chosen when the automatic configuration should be performed and changes merged into the current network model.

5.3.1 Topology

Topology discovery is a method of scanning a local network and determining the logical topology of that network. Topology discovery is important because the network may have devices hidden in many different logical segments of the network. There are a number of different techniques that can be used to discover the local network topology; some techniques work better than others and will reveal a more complete picture. One such technique to provide a full picture can be found in [3]. By discovering all network segments and finding the entire network we can ensure that monitoring black holes are not introduced to our network monitoring by leaving out segments from our network model.

5.3.2 Devices

Once we have a network topology, the next stage of the automatic configuration system is device discovery. This step is used to find devices on the individual segments of the network that was revealed by the topology discovery. This will result in a list of devices that are to be monitored. There are a number of different techniques available for scanning devices on a given network segment. Tools such as Network Mapper (Nmap) [13] implement some of these techniques already.

5.3.3 Services

Service discovery is similar to device discovery. When we have a list of devices to monitor each device can then be independently probed to determine the services that it hosts. This can be as simple as doing a scan of open ports on a machine and matching these against the well known ports that common services use. Open ports can also be queried for a banner to see what service is listening on that port. There are many tools that will do this such as Nessus [2] and the process is well understood.

5.4 Data Metric Relationships

Earlier we mentioned that it often requires expertise to identify the root cause of a fault. This section will look at a method that simplifies the output from a NMS and makes it more accessible to users.

We can define relationships, either automatically or manually, between data metrics that influence each other. This allows us to build smarter dashboards by grouping together related information to make it easier to find the root cause of a fault. For example, using these relationships to draw related graphs next to each other will in turn speed up diagnosis of faults and enable a user of the system to gain more information from a single screen then they otherwise would.

Chapter 6

Implementation

The initial evaluation of existing NMSs has demonstrated that no general purpose NMS currently exists that meets all of the requirements for this project. As a result, new methodologies and technologies are necessary to build a unified and refreshed NMS.

Furthermore, this project aims to produce a system that we will call a Dynamic Network Monitoring System, where the emphasis is on dynamic, rather than statically, configured network models. Many key design decisions have been made with this goal in mind.

DNMS has been designed to be a modular system for two reasons. First, features can be implemented incrementally while maintaining a working system at each development stage. This is important because building a feature complete NMS is outside the scope of this project due to time constraints. The second reason was to allow users to extend the feature set and tailor the NMS for their particular needs and to enable improved versions of modules, such as the data collector module, to be substituted without having to keep both loaded in the program at the same time. The modularity also helps reduce software bloat because the running version of the software only needs to enable the features required to monitor a particular network.

Another key design principal employed in the design of DNMS is to limit the amount of configuration that is required for the system to function. The minimum set of configuration required to monitor the network completely is stored. Anything not required is considered non-essential. This is implemented by attempting automatic configuration where possible and having sensible defaults pre-configured for use if automatic configuration fails. The user can override

these defaults if needed. This approach limits the amount of configuration that is required to run DNMS and reduce the likelihood of mistakes. Also because the configuration is easy to maintain, it is more likely to be up to date.

DNMS is written in Python as a web application built using the Pyramid [14] web framework. Figure 6.1 shows the overall system architecture and how the modules interact. The rest of this chapter looks at the different components of DNMS and the existing modules which provide the base functionality.

The biggest challenge faced in building DNMS in the Pyramid framework was understanding how the framework expected modules to be structured and keeping with the design pattern used by the framework. Some of the DNMS modules required some redesigning to coexist inside the framework and retain the core design goals of DNMS. Care also needs to be taken with variable consistency in Pyramid, as it is difficult to ensure a consistent value for a global shared variable between all requests to DNMS. Pyramid offers implementations of reliable persistent state sharing between threads that can be used instead.

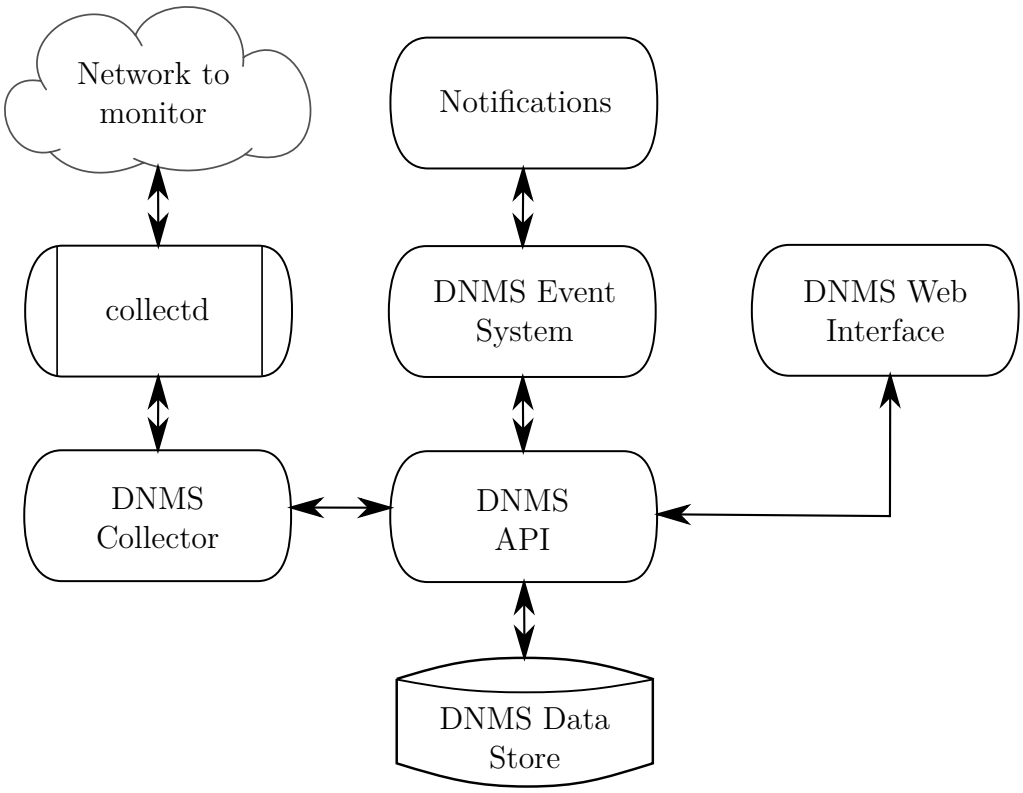

Figure 6.1: System architecture diagram for DNMS showing the flow of data through the system

6.1 DNMS Collector

The collector is the only component that is not directly integrated with the rest of DNMS. A data collector is a complex system that must handle many different tasks efficiently, such as collecting a large quantity of data from a device quickly without impacting the performance of that machine. It must provide a method to communicate with devices that is fast and efficient. Also, the collector must provide a robust caching layer because, as pointed out in a paper evaluating the performance of SNMP [29], the most expensive operation in a poll is the time taken to produce the data values. Caching reduces the impact on the device that is providing the data if the device is polled regularly. For simplicity, a pre-existing collection system, collectd [8], was adopted to be the data collector for DNMS.

The collectd architecture uses a hierarchical structure. A number of collectd instances can be deployed in a tiered configuration, as shown in Figure 6.2. In this example there is a central collectd instance which receives data from intermediary collectd instances. These instances collect data from a number of servers in a datacentre, which could also be running collectd locally, or the data can be acquired using SNMP.

The advantages of using collectd is that it natively supports many different collection approaches, including both push and pull data collection. It also supports pushing collected data to Multicast groups. This allows devices to export data to a Multicast address without having to be configured with the IP address of the collection server on the local network. The collectd instance simply exports data to a Multicast address and one or more collectd instances can be subscribed to this multicast data and collect all data pushed to this address.

The DNMS collector itself is implemented as a collectd plugin. The plugin was written in Python and uses the collectd Application Programming Interface (API) to export all of the collected data to the DNMS API for storage. The DNMS collector uses the built in collectd type definitions to determine the data type for a given measurement as well as the upper and lower bounds of the data metric. This reduces the amount of configuration related to the monitoring metrics. The DNMS collector module will also normalise any counter values into a rate, which is the change in a counter rather than the raw count. A counter value counts from zero up to the maximum value for an integer before

wrapping and resuming counting from zero again. Trends can be hidden and difficult to find in a graph of a counter value, but trends are easier to identify in a graph of the change of that counter.

6.2 DNMS Data Store

The DNMS data store is a SQL database running on a RDBMS. The core database logic code is database agnostic because it is based on SQLAlchemy [1], a Python SQL abstraction layer that is used throughout DNMS. As a result, the user can select the appropriate RDBMS for their needs. For example, the user may choose a simple RDBMS like SQLite [10], which requires little setup. For larger deployments, a more complex RDBMS with better performance may be required, such as MySQL [25] and PostgreSQL [28]. SQLite databases were used during the development of DNMS, but a simple configuration change allows another RDBMS to be used.

A SQL database was selected for this project as it allows the storage of many different data types, such as strings, integers and decimals, and has the ability to store and query a large number of entries efficiently. Another advantage of

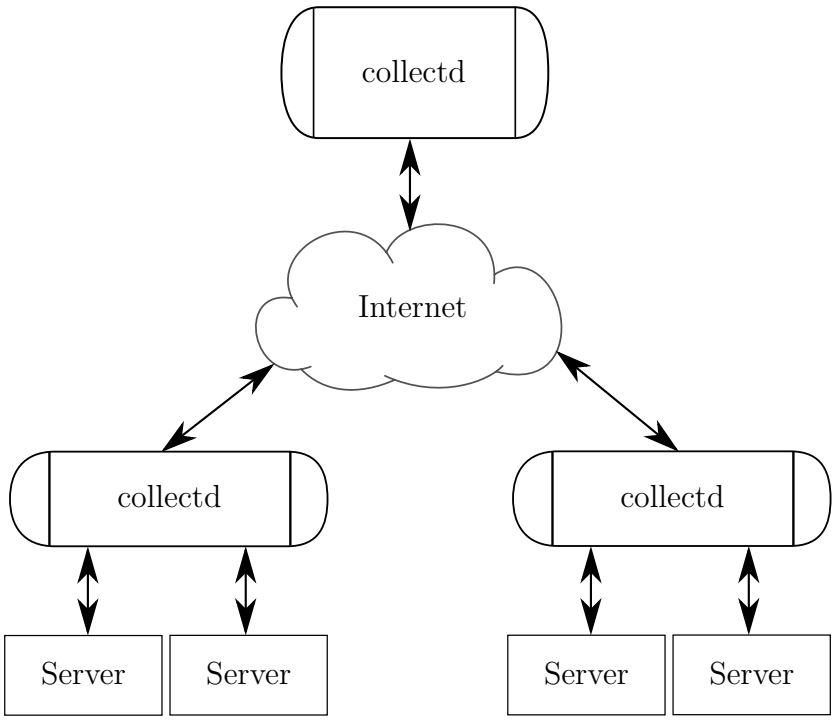

Figure 6.2: Diagram showing a collectd hierarchy monitoring two remote datacentres over the internet

using a SQL database is that all of the DNMS data can be stored in a single format instead of having separate storage formats for time-series data, events and configuration. By unifying the monitoring data, the system becomes less complex and easier to maintain. SQL itself is also a good basis for building rich modules because it is a powerful language that can be used to build powerful queries to analyse stored data and includes many built in functions such as aggregation. SQL databases are very mature; a lot of research and time has been spent optimising RDBMSs over the past four decades.

Challenges faced in SQLalchemy mostly surrounded discovering how the abstraction layered worked. DNMS tries to do as much data handling as possible in the SQL engine and some time was spent making SQLAlchemy perform these more complex queries. DNMS also utilises a feature of SQLAlchemy to automatically generate databases on the fly, but it required significant time and effort to configure SQLAlchemy to generate the databases automatically.

6.3 DNMS API

The DNMS API is the interface between other DNMS components and the data store. Data retrieval and data updates both utilise the API, providing a consistent interface for both operations.

The API itself is a RESTful API. REpresentational State Transfer (REST) is a design pattern used in web applications to represent objects transfered between a client and server. In DNMS, an object is a server or a data measurement. In DNMS, object interactions are performed by sending messages using the JavaScript Object Notation (JSON) format. JSON was chosen for this project because the Python JSON library is full featured and easy to use. JSON data also has the advantage of being human-readable which is helpful when debugging. The DNMS API has been built using the cornice [15] library for Pyramid which provides Python decorators which can be used to quickly develop RESTful APIs.

In REST, the HTTP vocabulary can be used to interact with these objects. The vocabulary relevant to this project includes:

- GET - Used for retrieving an object
- POST - Used for adding or modifying an object

- DELETE - Used for removing information

Challenges faced with building the API were mostly related to understanding Cornice. Code examples using Cornice showed it was the appropriate library to use for this project as it greatly simplified the code required to implement the API. However documentation is very limited for this library and it took some time to find the features of Cornice required to implement the API in the way DNMS required.

6.4 DNMS Event System

A simple event system is included with DNMS that employs thresholds for data points. The event system uses SQL queries to implement the thresholding and toggling of states. When a collected data point has a value greater than the configured threshold, an event is triggered and stored in the SQL database in the unacknowledged and active state. The event remains in this state until the data point drops below the threshold again and the event is changed to the inactive state. This ensures there is a history of events for later analysis. If there is a known problem that cannot be fixed, the event can be acknowledged but kept in the active state. This event will no longer be shown as an active alert on the web interface.

6.5 DNMS Web Interface

The DNMS web interface is primarily implemented as a proof-of-concept at this stage. It does not aim to be feature complete but demonstrates the existing system features. The web interface is written in HTML5 to ensure it will be widely supported on new web browser platforms and mobile devices.

The web interface provides a simple dashboard that shows all current active and unacknowledged events, this is shown in Appendix A.1. The dashboard allows the user to acknowledge or investigate the events further. The web interface also allows a user to explore monitoring data by viewing graphs of all data points stored by the data store. The graphs are highly interactive JavaScript graphs which are built using the flotr2 [11] JavaScript graphing library. Data for the graphs is fetched using the API and is automatically updated and redrawn regularly so that new data is presented to the user without

the web page having to be refreshed, this is shown in Appendix B.1.

There is also a basic implementation of relationships between data metrics. Currently, there is a static definition of data metrics that may impact each other and this definition list is consulted each time a graph is drawn. Related graphs are also drawn on the same web page in case they reveal the root cause of a particular event or alert. This system could be extended to do automatic correlation tests between graphs of data points to find related data metrics automatically and better populate the relationship list.

6.6 Testing & Development

Throughout this project, development and testing was performed on a flat IP network consisting of a small number of servers. Each server was running production services for a network research group. Data metrics are collected from each server using SNMP. The network was also monitored by Cacti and Icinga, ensuring that the operation of DNMS could be validated and compared.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

This project has shown the benefits and established how necessary network monitoring is on a network to properly administer and maintain it. Without monitoring, a network is a black hole and faults can go unnoticed for extended periods of time.

This project has shown that the most popular NMSs that are used today to monitor networks have a number of limitations and issues that prevent them from providing full testing coverage and detecting every fault. We have shown the major issues can be solved by using technology available today that was not available when these systems were originally developed.

Improved techniques for building a better NMS were devised and investigated. A new NMS was built from the ground up using new technologies. Some of the new techniques devised were integrated into the new NMS to improve usability and usefulness of a NMS.

7.2 Contributions

This project has made a number of contributions to the field of network monitoring. This report introduces a number of new techniques to the field that NMSs could use to improve configuration and improve the effectiveness of network monitoring.

A modular NMS, built out of new technologies, was developed during the course of this project. This NMS was built using a clean-slate approach and

used to implement the new methods investigated.

7.3 Future Work

NMSs are very large systems and to compete with a fully featured NMS would take a considerable time to develop the full set of features. DNMS has been implemented as the framework to build a feature complete NMS on top of. A number of the important core features to a NMS have been implemented as basic modules in DNMS. This section discusses the future work required to extend DNMS into a fully featured general purpose NMS.

7.3.1 Testing & Performance Tuning

DNMS was written to focus on implementing features over having good performance. DNMS still needs to have a large amount of testing carried out on its performance and scalability. A NMS needs to have good performance with a large number of devices to support large scale networks. Once performance problems are identified, if any, these can be tuned and optimised.

7.3.2 Dashboard

The dashboard, which is currently a static page showing current alerts needs a system behind it that allows it to be user configurable. The dashboard should be dynamically configurable to show important metrics and graphs chosen by the user. This level of customisation is necessary to make this system useful in all cases and monitoring scenarios.

7.3.3 Implement Modules

The modules implemented in this project offer a base set of features that a NMS might need. These are some good ideas for other modules that could be implemented to add to the feature set of DNMS and keep in with the goals of the project as a whole.

Service Polling

Currently DNMS only collects data pushed to it through the API, the is no active polling carried out. A service polling module would take the configured hosts it knows about and regularly poll them to make sure each device is

operational and all services are responding correctly. Each service check could be another module to keep bloat down and make it easy to implement a service check module without understanding the rest of the system. When a service check fails it can simply register an event with the event module and that can handle the notification and logging.

Automatic Configuration

A module that provides automatic configuration of devices and services to monitor through automatic discovery techniques would be an important module to implement to fully complete the Dynamic portion of the Dynamic Network Monitoring System. The current system relies on data showing up from devices and that they are correctly configured, as mentioned in the previous section no active polling occurs to check that devices are operational and running correctly and that our network is entirely operational. An automatic configuration module would make a good pair with the service polling module.

Notifications

The system could be easily extended to have a notification module that hooks into the event system so that when an event is triggered a notification is also generated and sent to the appropriate person. A notification system would also benefit from being modular and different notification modules could be turned on and off. For example notifications could be made over email, Short Message Service (SMS), pager or Instant Message (IM). Notification systems can range from simple systems that generate one notification per event that goes to everybody on a notification list or they can be very complex systems that try to limit the number of notifications sent about a given type of event so that services that are in the flapping state do not constantly alert somebody. More complex notification systems will also take into account work schedules allowing the on call hours of a person to be configurable so that notifications go to the right person at the right time. The point here being we can start with a simple module and swap it out with more complex ones depending on the use case.

Glossary

C a compiled procedural statically typed programming language. 14

Encapsulated PostScript (EPS) a more restricted form of a PostScript document that is intended for describing graphics. 10

Extensible Markup Language (XML) a markup language that defines rules that allows the construction of rich documents. XML is extensible allowing it to represent many different document and data formats. 14

HTML5 a markup language used for presenting content for the web. HTML5 is the current version of the HTML standard and seeks to improve multimedia and mobile support of previous standards among many other things. 28

Internet Control Message Protocol (ICMP) one of the underlying protocols used on the internet to send control messages such as error messages and echo packets. 12

JavaScript an interpreted dynamically typed scripting language that is the *de facto* scripting language used on the web because of its prevalence in web browsers. 28

JavaScript Object Notation (JSON) a plain-text human-readable data exchange format and popular for serialising and transmitting objects over a network. Like the name suggests it is based on the JavaScript format for representing objects and can be used to represent lists, associative arrays. 27

Lua a lightweight interpreted dynamically typed scripting language. Lua supports multiple programming paradigms and can be written in a procedural, prototype-based or object-orientated style. 14

Multicast a form of communication on a network where a message or stream is sent to a group of devices simultaneously with a single transmission. 25

Portable Document Format (PDF) a file format used for storing documents that are intended to be rendered the same independent of the environment used to render the file. 10

Portable Network Graphics (PNG) a bitmap image format that employs lossless compression to keep images small. 10

PostScript a programming language commonly used in desktop publishing and printing. 33

Python an interpreted dynamically typed cross-platform programming language. Python has a comprehensive standard library and doesn't lock the programmer into a single programming paradigm supporting object-orientated, imperative, functional, procedural and reflective styles. 13, 24–27

Redundant Array of Independent Disks (RAID) a method of combining multiple hard disk drives into a single storage volume of increased size and redundancy provided by either storing data multiple times or storing parity bits. There are multiple levels of RAID that provide different levels of speed and redundancy. 19

Relational Database Management System (RDBMS) a specific form of database management system that is based on the relational database model. 13, 17, 20, 26, 27, 34

RESTful an API is described as RESTful if it follows the REST design pattern. 27

Scalable Vector Graphics (SVG) a vector image format that uses an XML based file format to describe the vector graphics. 10

Standard Output (stdout) the output stream from a program where output data is written to. 12

Structured Query Language (SQL) a programming language designed specifically for managing data in a RDBMS. 12, 20, 26–28

References

[1] Michael Bayer. Sqlalchemy - the database toolkit for python. http://www.sqlalchemy.org. Accessed October 17, 2012.

[2] Jay Beale, Renaud Deraison, Haroon Meer, Roelof Temmingh, and Charl Van Der Walt. Service detection. In *Nessus Network Auditing*, page 248. Syngress Publishing, 2004.

[3] Y. Bejerano, Y. Breitbart, M. Garofalakis, and Rajeev Rastogi. Physical topology discovery for large multisubnet networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 1, pages 342 – 352 vol.1, march-3 april 2003.

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp). RFC 1157, RFC Editor, http://www.ietf.org/rfc/rfc1157.txt, May 1990.

[5] Chris Davis. Graphite - scalable realtime graphing. http://graphite.wikidot.com. Accessed October 17, 2012.

[6] Bernd Erk. Nagios is forked: Icinga is unleashed. http://www.icinga.org/2009/05/06/announcing-icinga/, May 2009.

[7] Stephen Few. What is a dashboard? In *Information Dashboard Design: The Effective Visual Communication of Data*, page 34. O'Reilly Media, Inc, 2006.

[8] Florian Forster. collectd – the system statistics collection daemon. http://collectd.org. Accessed October 17, 2012.

[9] Ethan Galstad. Nagios history. http://www.nagios.org/about/history. Accessed September 26, 2012.

[10] Richard Hipp. Sqlite. http://www.sqlite.org. Accessed October 17, 2012.

[11] Humble Software. Flotr2. http://www.humblesoftware.com/flotr2/. Accessed October 17, 2012.

[12] Information Sciences Institute, University of Southern California. Internet protocol. RFC 791, RFC Editor, http://www.ietf.org/rfc/rfc791.txt, September 1981.

[13] Gordon Fyodor Lyon. Host discovery (ping scanning). In *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*, page 47. Insecure, 2009.

[14] Chris McDonough and Ben Bangert. Pyramid. http://www.pylonsproject.org. Accessed October 17, 2012.

[15] Mozilla Project. Cornice. http://github.com/mozilla-services/cornice. Accessed October 17, 2012.

[16] Nagios Enterprises. Nagios - the industry standard in IT infrastructure monitoring. http://www.nagios.org. Accessed October 17, 2012.

[17] Nagios Plugins Development Team. Nagios plugins - the home of the official plugins. http://nagiosplugins.org. Accessed October 17, 2012.

[18] Tobias Oetiker. Mrtg - the multi router traffic grapher. http://oss.oetiker.ch/mrtg/. Accessed October 17, 2012.

[19] Tobias Oetiker. Rrdtool. http://oss.oetiker.ch/rrdtool/. Accessed October 2, 2012.

[20] Tobias Oetiker. Rrdtool. http://oss.oetiker.ch/rrdtool/. Accessed October 17, 2012.

[21] Tobias Oetiker. Smokeping. http://oss.oetiker.ch/smokeping/. Accessed October 17, 2012.

[22] Tobias Oetiker. Rrdtool 1.0.0. http://lists.oetiker.ch/pipermail/rrd-announce/1999-July/000007.html, July 1999.

[23] OmniTI Labs. Reconnoiter. http://labs.omniti.com/labs/reconnoiter. Accessed October 17, 2012.

[24] OmniTI Labs. Reconnoiter - goals. http://labs.omniti.com/labs/reconnoiter/wiki/Goals. Accessed October 18, 2012.

[25] Oracle Corporation. Mysql - the world's most popular open source database. http://www.mysql.com. Accessed October 17, 2012.

[26] Colin Pattinson. A study of the behaviour of the simple network management protocol. In Olivier Festor and Aiko Pras, editors, *DSOM*, pages 305–314. INRIA, Rocquencourt, France, 2001.

[27] Avery Pennarun, Bill Allombert, and Petter Reinholdtsen. Debian popularity contest. http://popcon.debian.org. Accessed October 17, 2012.

[28] PostgreSQL Global Development Group. Postgresql - the world's most advanced open source database. http://www.postgresql.org. Accessed October 17, 2012.

[29] Aiko Pras, Thomas Drevers, Remco Meent van de, and Dick Quartel. Comparing the performance of snmp and web services-based management. *IEEE Transactions on Network and Service Management*, 1(2):72–82, December 2004.

[30] The Cacti Group, Inc. Cacti: The complete rrdtool-based graphing solution. http://www.cacti.net. Accessed October 17, 2012.

[31] The Icinga Project. Icinga: Open source monitoring. http://www.icinga.org. Accessed October 17, 2012.

[32] UNINETT. Network administration visualized. http://metanav.uninett.no. Accessed October 17, 2012.

Appendix A

Dashboard of DNMS

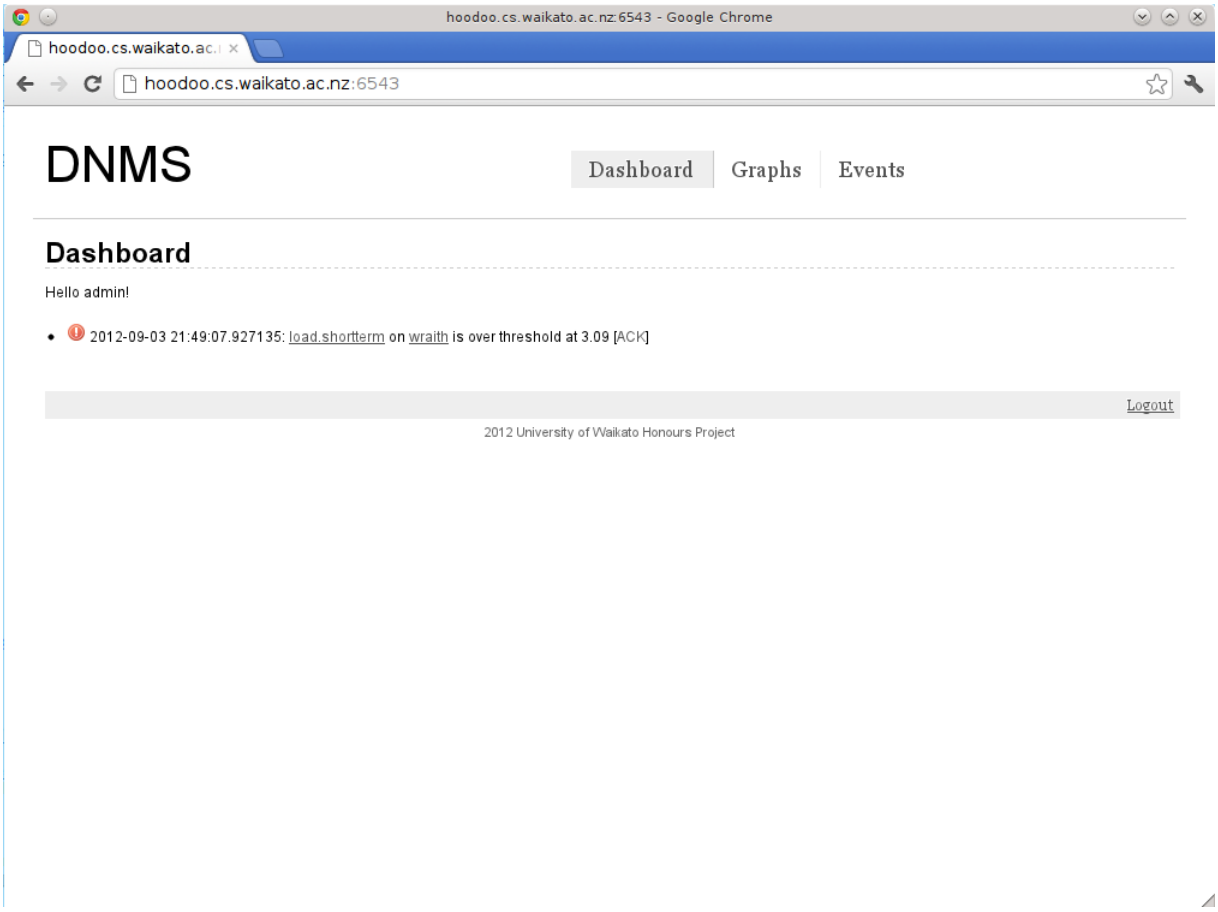

Figure A.1: The dashboard interface of DNMS showing a single alert

Appendix B

Graph interface of DNMS

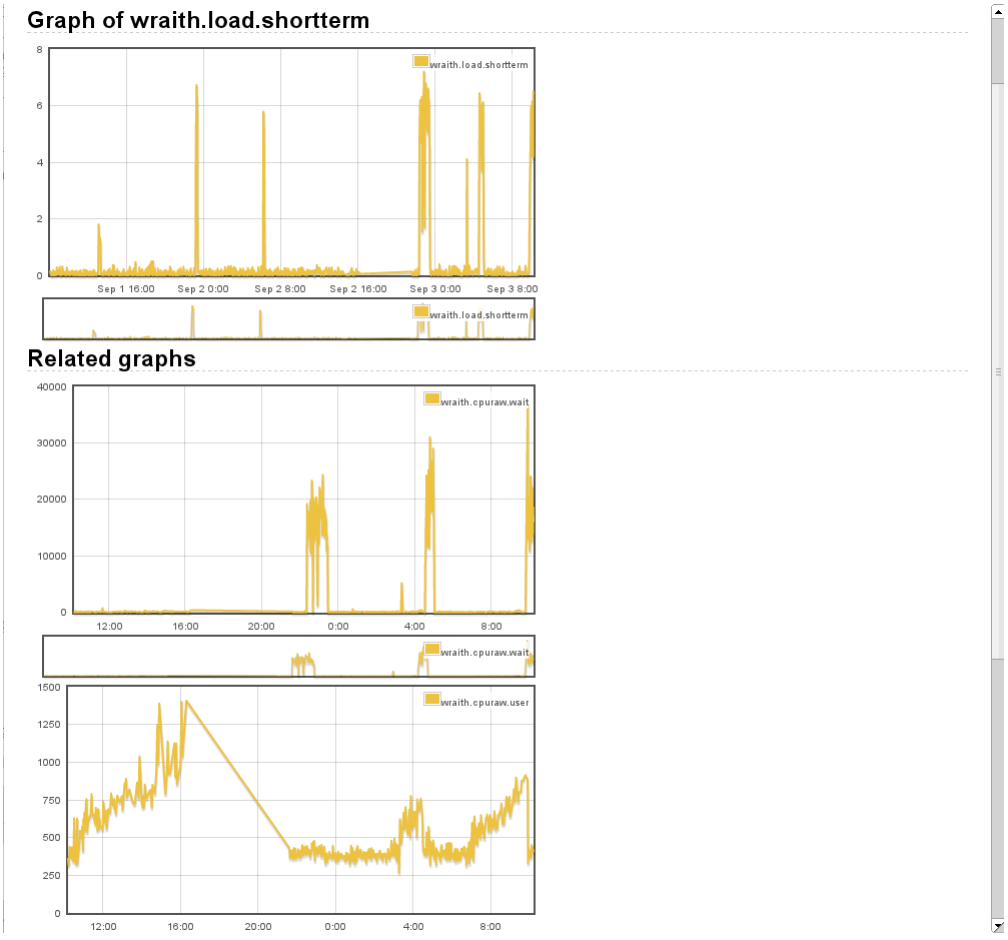

Figure B.1: A sample graph page in DNMS showing short term load for a server and related graphs