Prophasis - An IT Infrastructure Monitoring Solution

Cameron Gray

Fourth Year Project Report
School of Informatics
University of Edinburgh
2016

Abstract

Prophasis is an IT infrastructure monitoring system that is designed to suit small to medium size businesses where a system needs to be intuitive to manage. Management of the entire system can therefore be handled from a single, responsive web interface. It is also suitable as a one-stop tool with support for both time series monitoring in addition to real time alerting. Traditionally two different tools would be needed to gain this level of monitoring.

Table of Contents

1 Introduction **7**
- 1.1 Background . . . 7
- 1.2 Prior Work . . . 7
  - 1.2.1 Time series monitoring and real time alerting . . . 8
  - 1.2.2 Support for Custom Metrics . . . 8
  - 1.2.3 Classification Threshold Definitions . . . 8
  - 1.2.4 Code/Configuration Delivery to Nodes . . . 9
  - 1.2.5 How Dependencies are Handled . . . 9
- 1.3 Improvements . . . 9
  - 1.3.1 Configuration Management . . . 10
  - 1.3.2 Time Series Monitoring & Real Time Alerting . . . 10
  - 1.3.3 Expandability . . . 10

2 Design **11**
- 2.1 Technology Choice . . . 11
- 2.2 System Structure . . . 12
  - 2.2.1 Agent . . . 12
  - 2.2.2 Core . . . 13
  - 2.2.3 Web Interface . . . 17
- 2.3 Monitoring Methodology . . . 17
  - 2.3.1 Host Management . . . 17
  - 2.3.2 Plugins . . . 17
  - 2.3.3 Checks . . . 18
  - 2.3.4 Schedules . . . 18
  - 2.3.5 Services . . . 18
  - 2.3.6 Alerts . . . 19
  - 2.3.7 Classification . . . 19
- 2.4 User Interface . . . 19
- 2.5 Determining the Overall Health of a Host or Service . . . 20

3 Implementation **21**
- 3.1 Technologies . . . 21
  - 3.1.1 Flask Web Framework . . . 21
  - 3.1.2 Tornado HTTP Server . . . 21
  - 3.1.3 SQLAlchemy ORM . . . 22
  - 3.1.4 Lupa . . . 22

3.1.5 PostgreSQL Database 22
3.1.6 Appdirs 22
3.1.7 AdminLTE & Bootstrap 22
3.1.8 Handlebars.js 22
3.2 Plugin Structure 23
3.2.1 Plugin Logic 24
3.3 Database 25
3.4 Common Module 27
3.5 Core 27
3.5.1 Dispatcher 27
3.5.2 Classification 27
3.6 Agent 28
3.7 Web Interface 29
3.7.1 Configuration 29
3.7.2 Reporting 34

4 Testing **37**
4.1 Unit Testing 37
4.2 Mock Agent 37
4.3 Use in the Field 39
4.3.1 The Tardis Project 39
4.3.2 Lynchpin Analytics Limited 40

5 Evaluation **43**
5.1 Future Work 44

6 Conclusion **47**

Chapter 1

Introduction

1.1 Background

In recent years, almost all businesses have been expanding their IT infrastructure to handle the modern demand for IT systems. As these systems grow and become increasingly important for business operation it is crucial that they are sufficiently monitored to prevent faults and periods of downtime going unnoticed. There is already a large market of tools for monitoring IT systems however they are designed for use on massive scale networks managed by teams of specialised systems administrators. They are therefore complicated to set up and manage and multiple tools are often required to gain a suitable level of monitoring.

For example, tools generally either fall into the category of real time alerting (i.e. telling someone when something breaks) and time series monitoring (i.e. capturing data about the performance of systems and presenting graphs and statistics based on it), there is a large gap in the market for tools that provide both of these in one package. This reduces the time required to manage the system as it eliminates the need to set up and configure two completely separate tools.

These tools are also generally managed and configured through various configuration files split across different machines on the network. This means that in order to efficiently use these tools a configuration management system such as Puppet must be used. In a small business with limited IT resources, a completely self contained system is often preferable.

1.2 Prior Work

This section will review current IT infrastructure monitoring systems and evaluate them on several points as follows:

- Support for timeseries monitoring and real time alerting
- How they can be configured to monitor custom metrics

- How are alert thresholds defined
- How configuration and custom code is delivered to nodes (if required)
- How the user configures the system
- How dependencies are handled

1.2.1 Time series monitoring and real time alerting

This is a section where there appears to be a lack of tools that support both time series monitoring as well as real time alerting in any reasonable capacity. Nagios and Icinga2 are mainly focused on real time alerting so do not have much in the way of support for time series monitoring. Additional plugins are available to perform basic graphing functionality however historical data cannot be used to make decisions when it comes to determining the overall health of a host. In contrast, Munin is designed primarily as a tool for time series monitoring. Therefore it has extensive functionality for graphing collected data but has no built in support for alerting about faults or working out the health of a system.

This identifies an issue where there is no single tool that performs well for both real time alerting as well as time series monitoring. This means that in order to gain both of these, two separate tools (e.g. Nagios and Munin) need to be installed, configured and maintained completely separately.

1.2.2 Support for Custom Metrics

Nagios, Icinga 2 and Munin all operate in similar ways here through the use of simple scripts that can be written in any language. Both Nagios and Icinga 2 execute the shell script and use the exit code to determine the health of the output (e.g. Exit code 0 means "OK" and exit code 2 means "CRITICAL"). They also read data from stdout which can contain more information about the status of the check. Munin plugins are similar where the field name and a numerical value are written to stdout and then picked up by the plugin. Plugins must also, when passed "config" as a command line argument, print out information such as the title of the graph.

In this area, current tools are all very much the same in the sense that they execute scripts and parse the output. This can be quite fragile as a badly written script could fairly easily output data in the wrong format. Using exit codes to output the overall health of a check is also very restrictive as described in subsection 1.2.3.

1.2.3 Classification Threshold Definitions

Both Nagios and Icinga2 classify the result of a check within the plugin itself by the plugin exiting with the appropriate exit code to define the result. This means

that the thresholds are also generally stored inside the plugin script. Because Munin does not support real time alerting, there is no support for classifying data.

This is a clear issue with current systems, all classification is performed on remote machines meaning that the logic used for this along with any threshold values are located on every machine in the network making it difficult to update. This also means that classifications can only use the current data collected and can't easily use historical data when deciding on a classification.

1.2.4 Code/Configuration Delivery to Nodes

Nagios, Icinga2 and Munin are configured through several different configuration files and scripts stored on the disks of the server running the monitoring as well as the hosts being monitored. There is no built in functionality in any of these tools for distributing these files to their respective machines. For this, separate configuration management tools such as Puppet or Ansible would need to be used.

This is a clear area for improvement, in a small business environment with a limited number of machines, each with their own, separate role, often configuration management tools are not deployed. Deploying an existing tool in an environment like this would be time consuming as configuration would need to be manually copied between hosts and editing the configuration later on would be a manual process of logging into every machine and changing the configuration files manually which always risks creating discrepancies between hosts if one gets missed during an update or updated incorrectly.

1.2.5 How Dependencies are Handled

Nagios and Icinga 2 both handle dependencies as a rigid tree where hosts can be set as dependent on other hosts. There is however no support for redundancy where the availability of a service can be dependant on one of several hosts being available. This is increasingly common in modern environments which often include large amounts of redundancy and failover capacity. Munin has no dependency functionality due to its lack of real time alerting.

This is also a clear area for improvement, in a modern environment it is important for a monitoring system to be able to understand redundancy.

1.3 Improvements

Prophasis is designed for use in a small to medium business with limited IT resources. They may have a small IT team with limited resources or may not even have a dedicated IT team at all, instead relying on one or two employees

in other roles who manage the business's IT systems on the side of their regular jobs. Therefore the system needs to be quick to deploy and manage with a shallow learning curve. In order to use the system efficiently there should be no requirement for additional tooling to be deployed across the company.

1.3.1 Configuration Management

It should be possible to manage the configuration of the system from a single location. Prophasis therefore provides a responsive web interface where every aspect of the system's operation can be configured, Prophasis then handles distributing this configuration to all other machines in the system in the background. Custom code for plugins is handled in the same way; it is uploaded to the single management machine and is then automatically distributed to the appropriate remote machines when it is required.

1.3.2 Time Series Monitoring & Real Time Alerting

Prophasis provides both the ability to alert administrators in real time when a fault is discovered with the system alongside functionality to collect performance metrics over time and use this data to generate statistics about how the system has been performing. This time series data can be used to both investigate the cause of a failure in post-mortem investigations in addition to being able to be used to predict future failures by looking at trends in the collected data.

1.3.3 Expandability

It is important that a monitoring tool can be expanded to support the monitoring of custom hardware and software. An example of this would be hardware RAID cards. Getting the drive health from these types of devices can range from probing for SMART data all the way to communicating with the card over a serial port. It is therefore crucial that Prophasis can be easily expanded to support custom functionality such as this. Therefore Prophasis supports a system of custom "plugins" which can be written and uploaded to the monitoring server where they can then be configured to monitor machines. These plugins are designed to be self contained and to follow a well defined and documented structure. This provides scope for a plugin "marketplace" therefore eliminating the need for every user to implement custom monitoring code for the systems they are using.

Chapter 2

Design

2.1 Technology Choice

Python It was decided to use Python as the language of choice to implement Prophasis. This was chosen for many different reasons: it is available on practically all platforms and is usually bundled with most UNIX-like operating systems. It also abstracts a large amount of low level details away which both saves development time and prevents subtle, hard to find errors such as memory management issues. Python also has a huge library of available plugins along with a powerful and flexible package manager (pip) and package repository (PyPi). This allows external plugins to be used to provide functionality instead of requiring everything to be implemented from scratch. This saves development time and means that specialised, well tested and maintained code can be used to provide some of the functionality. Pulling in packages from PyPi instead of bundling external libraries with Prophasis ensures that any external libraries are kept up to date and prevents any licensing issues that could arise from bundling code from other sources alongside the Prophasis source code.

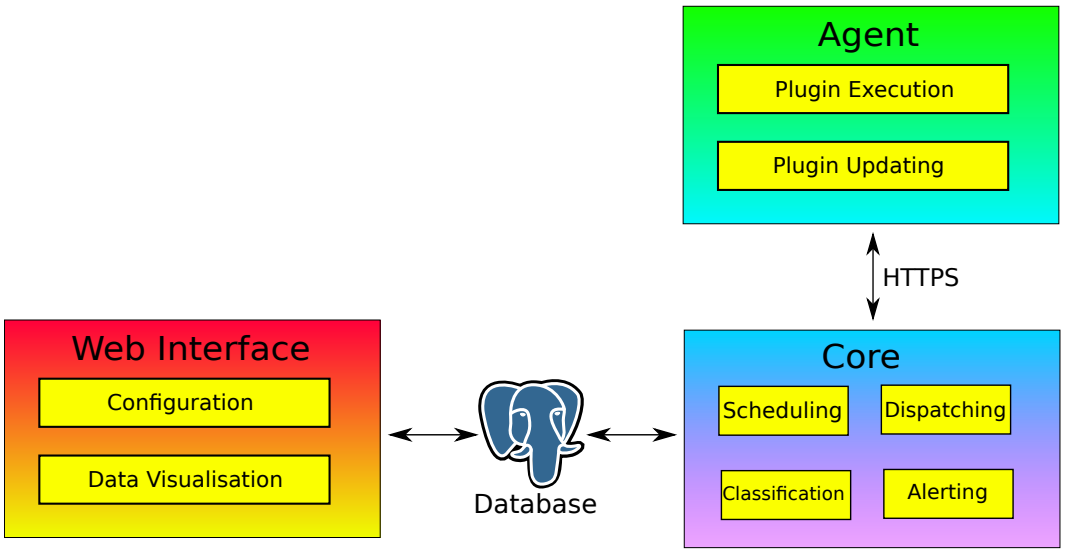
HTTP for Communication Protocol HTTP was chosen as a communication protocol for several reasons. It is well supported and understood with a large variety of server and client libraries available. Using a premade library has major benefits for development time, security and support. HTTPS also provides suitable encryption and certificate validation technologies to secure the system. HTTP is also widely permitted through firewalls and can be routed through web/reverse proxies. This means that Prophasis can be operated on most networks without causing problems with firewalling. HTTP also includes methods for transferring simple data as well as large binary files which is required for sending plugins to remote agents. Originally an attempt was made to use SSH to implement communication between the core and the agent. However, it was found that this introduced a huge amount of additional complexity on both the server and client, especially if the SSH server were to be embedded into the agent to keep it as a self contained application. This additional complexity not only made maintenance and development harder, it also increases the risk of errors being introduced. SSH was also more difficult to route through restricted net-

works. Another benefit of using HTTP is how many well documented HTTP libraries and servers there are available, this means that alternative agents can be implemented with ease versus SSH which would involve having to handle a custom wire protocol.

2.2 System Structure

The system is split up into three separate components; Web, Core and Agent. Web and core both share the same relational database allowing data to be shared between them. Figure 2.1 shows the individual components of the system and how they all interact.

Figure 2.1: Diagram showing the layout of the components of the system



2.2.1 Agent

The agent runs on every machine that is being monitored and provides an API for communication with the monitoring server. It listens on port 4048 (by default) and exposes an HTTPS API. This API includes methods to check the version of a plugin currently installed on the agent, a method to push updated plugins to the agent and another method to execute a plugin and to retrieve the value and message produced by it.

The agent's API is authenticated using a long, pre-shared key of which a salted hash is stored on the agent in a configuration file. Being hashed prevents users who may have access to read the configuration file (possibly through misconfiguration) from getting the key to be able to communicate with the agent.

2.2.2 Core

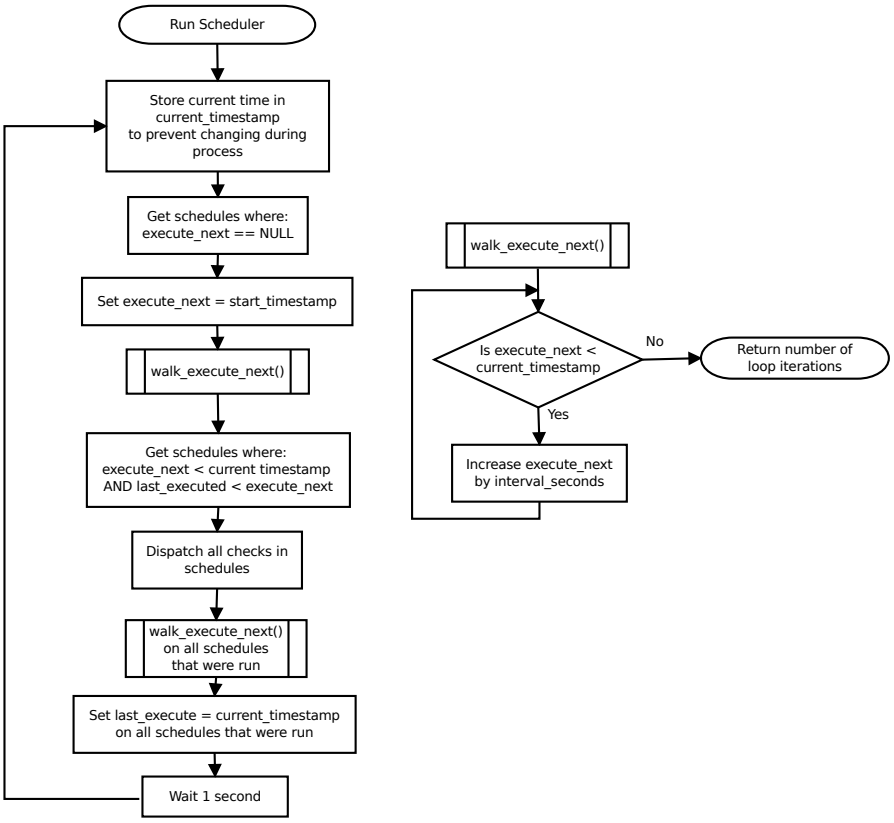
The core runs on the machine that is performing the monitoring, it has several different roles; scheduling, dispatching, classification and alerting.

2.2.2.1 Scheduling

The core is responsible for looking at the schedules stored in the database and executing the appropriate checks on the correct hosts at the correct time. There is a configuration value for "maximum lateness" that defines how late after its defined time slot a check can be executed. The core repeatedly checks the database looking at the intervals for each schedule along with the time at which a given schedule was last executed. If it decides that a schedule is due to be executed it passes this onto the dispatcher.

Figure 2.2 describes how the scheduler operates. Each schedule has a `start_timestamp` which is defined by the user when the schedule is created, an `interval` which is how often the schedule executes and a value for `execute_next` which is the timestamp that the schedule is next to be executed. When the scheduler starts up it first gets all schedules that do not have an `execute_next` value - These are schedules that have never run. It then calls `walk_execute_next` which is a simple algorithm that "fast forwards" the `execute_next` value until it reaches a timestamp that is in the future. It then retrieves any schedules that are due to be executed (`execute_next` is in the past) and executes them, it then calls `walk_execute_next` on each of these to set the `execute_next` value to the time that the schedule should be run again. The algorithm will then wait for 1 second before executing the process again.

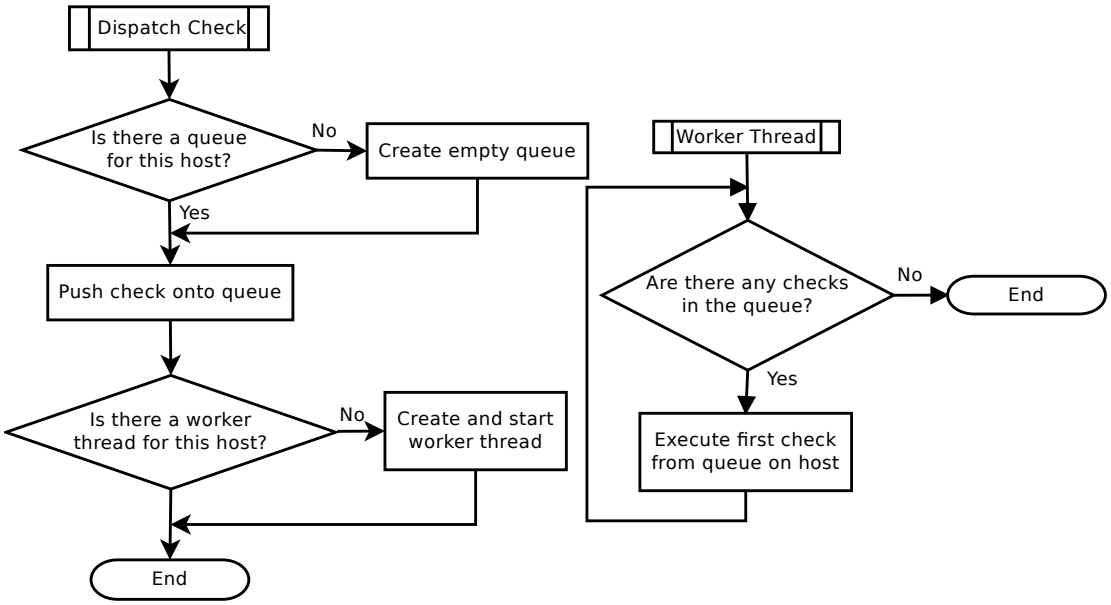
Figure 2.2: Flowchart describing the operation of the scheduler



2.2.2.2 Dispatching

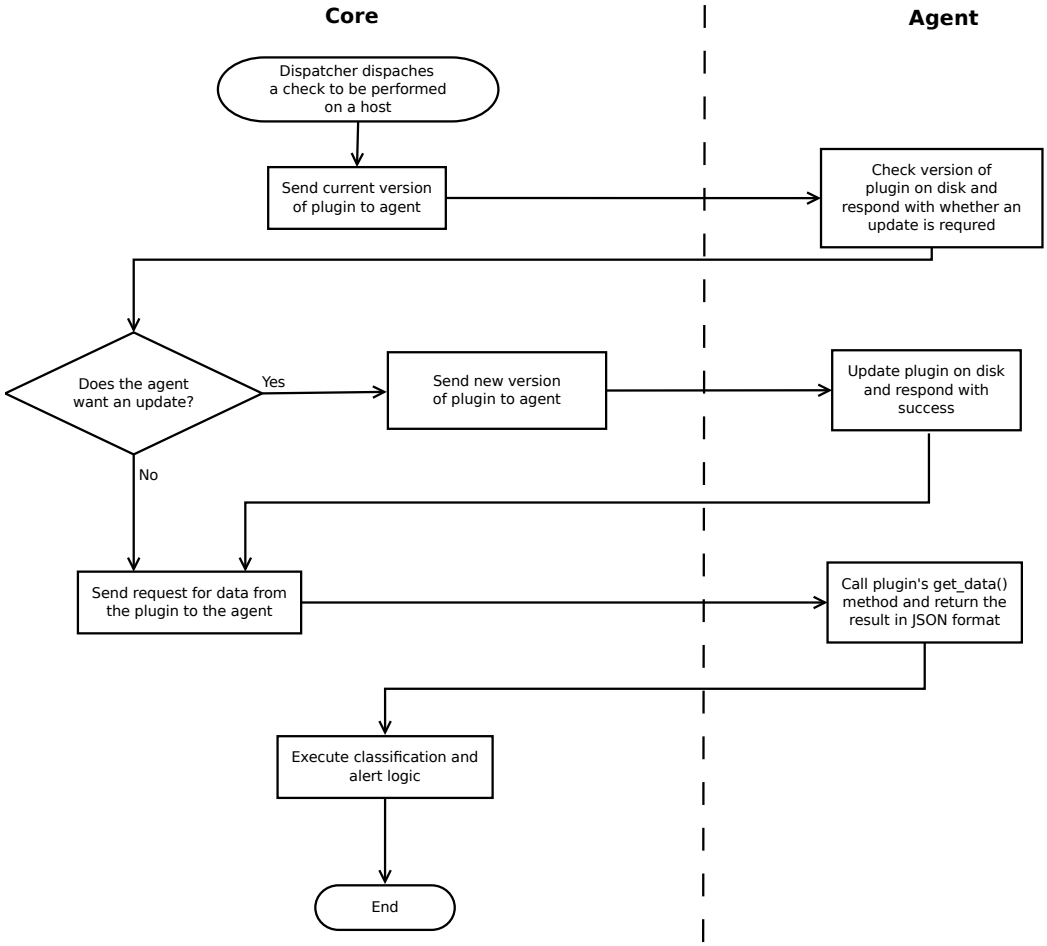
The dispatcher component of the core is responsible for issuing checks to agents when they are due to be run (as decided by the scheduler). Checks may take some time to execute so executing these all in series would all be impractical. The solution for this was for the dispatcher to spawn a process for each agent that it is currently executing checks for. Each process maintains a queue of plugins that are due to be executed and issues them to the agent in the order that they were dispatched. This way only one plugin can be executing on a given agent at any moment in time. This both prevents agents from becoming overwhelmed and means that plugin developers do not need to be concerned about other plugins interfering with their plugin. Figure 2.3 shows the operation of the dispatcher.

Figure 2.3: Flowchart describing the operation of the dispatcher



Communication Between Core and Agent When a check is dispatched, the core and agent communicate to first of all establish if the agent already has the correct version of the plugin due to be executed installed. If it does not then an update will be sent to the agent. Once this is done the core will then request the agent to execute the plugin and the data will be returned for classification. Figure 2.4 describes the communication between the core and the agent.

Figure 2.4: Flowchart describing the communication between the core and the agent when a check is dispatched



2.2.2.3 Classification

When data is collected from an agent, it needs to be classified as "ok", "major", "minor", "critical" or "unknown". Classification is performed by lua code that is stored alongside each plugin. When the core finishes collecting a from an agent it will retrieve the classification code for the plugin and execute it in a sandboxed lua runtime environment. The result of the classification is then stored in the database. The use of Lua code provides total flexibility, classifications can be as simple as comparing values to a threshold or could go as far as looking at previous historical values and classifying based on a trend in the data. Plugins define how many previous historical values they want to classify on (n), the plugin's classifier code is then executed with two Lua tables, one containing the previous n "values" stored and the other containing the previous n "messages".

2.2.2.4 Alerting

Once the core has classified a piece of data, the core now needs to work out whether it needs to send an alert or not. To do this it looks at the previous

state for the plugin and host along with the current state. It then looks through the database for alerts that match that state transition and are applicable to the host/plugin combination. If any alerts match it will call the alert module to send the alert out to inform the necessary people about the state change.

2.2.3 Web Interface

Prophasis provides a web interface for both configuring the system and viewing collected data. The interface is designed to be clear and easy to understand for users. It is also built to be fully responsive so will work correctly on mobile device without any loss of functionality. Designing it to be responsive was the aim from the outset as servers often fail when the administrator may not currently be in easy reach of a computer so will need to use a mobile device to investigate the issue.

The web interface will connect to the same database as the core, the web interface will update the configuration data stored in this database and will retrieve the data about hosts.

The web interface provides dashboards for visualisation of data collected from plugins. The collected data can then be displayed using graphs, tables, lists of messages or any other way that suits the type of data collected.

2.3 Monitoring Methodology

This section explains how Prophasis is laid out and introduces terminology used throughout the system.

2.3.1 Host Management

In Prophasis, a "host" refers to a single machine that is being monitored. To aid management and organisation, it is possible to organise hosts into "Host Groups." These can be comprised of hosts or other groups and can be used in place of hosts when defining services and checks. Hosts can be grouped in various ways such as their role (Webserver, VM Host), hardware configuration (Hardware RAID, Contains GPU), location (Edinburgh Office, Datacenter) or any other way that makes sense given the specific implementation.

2.3.2 Plugins

A "plugin" is a package that checks a single attribute of a system. For example a plugin could check the CPU load of a given machine or be used to check the health of all the hard drives in a system. Plugins are implemented as Python modules that implement a specific API, they can return both a "value" (a numerical representation of the data they collected) and/or a "message" (a string representation of the collected data). Plugins are then packaged in a .tar.gz

archive along with a manifest JSON file which contains information about the plugin for use when it is installed.

Plugins are automatically distributed from the core to remote machines and when executed by the agent the value and message are returned to the core for classification and storage in the database.

2.3.3 Checks

A "check" is a named set of hosts/host groups and a set of plugins to be executed on them. When a check is executed, all of the plugins specified in the check will be executed across all of the hosts specified in the check.

Checks allow logical grouping of various plugins. For example you may have a check for "Webserver Resource Usage" which will execute plugins to check the CPU load and memory across all hosts in the "Webservers" host group.

2.3.4 Schedules

A schedule defines when one or more checks are to be executed. Each schedule can contain multiple "intervals" which define when the check is run. An interval has a start date/time and a time delta that defines how regularly it is to be implemented.

2.3.5 Services

Services can be used to define a more fine grained representation of how the availability of a host affects the performance of the system overall. A service is defined in terms of "dependencies" and "redundancy groups". Each dependency represents a host that must be operational for the service to work. A redundancy group can contain multiple hosts where at least one must be operational for the service to work.

As an example, you may have a "Website" service that has a dependency on the single router but has a redundancy group containing multiple webservers. Therefore, if the router fails then the website will be marked as failed however if one of the webservers fails but at least one other webserver is still operational, the website service will only be marked as degraded.

The use of services provides a clearer view of the impact of a given failure on the functionality of the network. It also allows alerts to be set up so that they are only triggered when the service functionality is severely impacted and prevents alerts from being sent out for host failures that do not have a severe impact.

2.3.6 Alerts

In Prophasis, alerts are set up to communicate with a specific person/group of people in a certain situation. Alerts are defined in terms of to/from stage changes where an alert will be sent if the state of a host/service changes from one state to another (e.g. "ok" to "critical"). These can then be restricted to certain hosts, host groups, services, plugins and checks.

The system supports "alert modules" - Python modules that are used to send alerts using various methods. Examples of alert modules could be email, SMS or telephone call.

Multiple alerts can be set up to handle different severities of issues. For example, if a redundant service becomes degraded, a SMS or email message may be sufficient but if a service becomes critical it may be desirable to use a telephone call to gain quicker attention.

2.3.7 Classification

A classification defines how serious or important the data from a plugin is, for example, the CPU load being slightly higher than normal would probably be classed as a minor issue whereas a system not responding to checks from Prophasis would be likely to be classed as a critical issue. Table 2.1 lists all the different classifications ordered by severity.

Table 2.1: Different classifications that can be given to data ordered by severity

Severity	Classification	Notes
Most severe	Critical	
⋮	Major	
⋮	Minor	
⋮	Degraded	Only applies to services
⋮	Unknown	
⋮	Ok	
Least severe	No data	Plugin has never been executed on this host

2.4 User Interface

The nature of a monitoring system means that it will often need to be used from mobile devices when systems need to be checked on the move or out of hours. Therefore Prophasis's web interface must work correctly on both mobile devices and conventional desktops and laptops. It was therefore decided to build a responsive web interface that will scale automatically based on the screen size of the device without any loss of functionality.

In order to keep Prophasis as intuitive as possible, the user interface should contain clear explanations of different functionality as well as help text and detailed error messages.

2.5 Determining the Overall Health of a Host or Service

In Prophasis, the data returned by each plugin is classified individually. However, in order to display the collected data or to create alerts, we need to be able to decide on a general health classification for an entire machine or service based on the classifications for the individual plugins.

Determining health of a host In order to determine the health of a host we first need to find out what checks the host in question is part of. For this we query the database to get a list of all checks the host is assigned to and then for each host group that the host is part of, we get all checks that the group is assigned to. We then go through all of the checks that have been found to build a de-duplicated list of all plugins that are being executed on the host in question. Then, for each plugin, we get the result of the most recent check of the plugin on the host. Then, using the severity ordering in Table 2.1, we pick the most severe classification from the plugins and use this as the overall health of the host.

Determining the health of a service Determining the health of a service is somewhat less clear cut due to the use of dependencies and redundancy groups. In Prophasis we take all dependencies and redundancy groups in the service and get the health of each. We take the health of a redundancy group to be the least severe health out of all hosts in the group (unless the least severe is "ok" and there are non-"ok" hosts in which case the group is classed as degraded). For example, a redundancy group containing a host with health "minor" and a host with health "major" would be classed as "minor" and a redundancy group with a host of health "critical" and a host of health "ok" would be classed as "degraded". We then take the overall health of each redundancy group along with the health of all dependencies (which is simply the health of the host that the dependency is for) and then take the most severe of all of these. Severities are determined based on Table 2.1.

Chapter 3

Implementation

3.1 Technologies

The vast majority of the system is implemented in Python 3. This allows for a large variety of modules to be used during development. Python is also widely available on UNIX systems and is easy to install on machines where it is not included.

The Python "Virtual Environment" (Virtualenv) system is also extremely useful in this system. It allows all dependencies to be kept totally separate from the rest of the system, this is particularly important for the agent as it ensures that the agent cannot interfere with the Python environment of the systems it is running on.

3.1.1 Flask Web Framework

The agent and web interface are both built using the Flask web framework which handles routing URLs to the correct Python functions as well as handling request data and building the correct HTTP responses. The agent purely uses the routing and request/response functionality provided by Flask whereas the web interface also uses Flask's bundled templating engine, Jinja2, to render the HTML pages that are displayed to the user. The web interface also uses the Flask-Login package to provide login and user session management functionality.

3.1.2 Tornado HTTP Server

While Flask does provide a built in webserver, it is only designed for development use. In order to provide a production suitable web server for the agent. Tornado uses the uWSGI interface provided by Flask. Tornado was chosen as it can easily be integrated directly into the Flask application and therefore does not require any sort of external webserver to be installed/configured on the system.

3.1.3 SQLAlchemy ORM

All database functionality in Prophasis is handled through the SQLAlchemy ORM which abstracts the database into a set of Python classes (known as models). This not only reduces development time, it also reduces the likelihood of errors as all of the database queries are generated by the library rather than being handwritten. The SQLAlchemy models file can also be shared between both the web interface and the core preventing duplication of database query logic.

3.1.4 Lupa

Lupa provides an interface between the system's Lua runtime and Python. It is used by Prophasis to execute the user provided Lua code that is used to classify the results of checks. A sandboxed Lua environment is configured to prevent this user provided code from performing undesired operations. Sandboxing is covered further in section 3.5.2.1.

3.1.5 PostgreSQL Database

Prophasis has been officially developed to support PostgreSQL databases, however through the use of an ORM it is possible to easily move to different database platforms such as MySQL or Oracle. SQLAlchemy transparently handles the difference between different database platforms when generating its queries.

3.1.6 Appdirs

Prophasis has been designed to support a variety of different operating systems. Different platforms have different conventions for where an application should store configuration files and other application data. Appdirs is a Python library that works out the correct path for various types of files for the detected platform. This means that files are stored in the correct directory no matter if Prophasis is installed on Linux, BSD, Windows or an other supported platform.

3.1.7 AdminLTE & Bootstrap

AdminLTE is an open source theme for building administration dashboards and control panels. It is MIT licensed and is therefore compatible with the MIT licence Prophasis is built under. AdminLTE is built on top of Twitter's Bootstrap framework which is well supported with a large range of plugins available. AdminLTE and Bootstrap have extensive support for responsive user interfaces and therefore very little manual work needs to be done to make the user interface responsive to work well on both desktop and mobile devices.

3.1.8 Handlebars.js

Handlebars.js is a very lightweight Javascript templating engine. It is used in the frontend of the web interface for rendering portions of the page that are generated

after the page has loaded, for example alerts and items that are dynamically added to the DOM such as dependencies when editing services or intervals when editing schedules. This means that any HTML logic can be totally separated from the Javascript source code. Unfortunately Handlebars templates are slightly different to Jinja2 ones. For simply filling in placholders they are compatible with one another however, more advanced functionality such as conditionals and loops require slightly different syntax. Therefore care must be taken when rendering the same template with both Jinja2 and handlebars.

3.2 Plugin Structure

Plugins are packaged as gziped TAR archive containing several different files.

Manifest File All plugins must contain a file to store information about the plugin called `manifest.json`. Table 3.1 explains the different attributes of the manifest file and if they are required or not. The additional files described in the manifest file will be explained later in this section.

Table 3.1: The contents of a plugin's manifest file

Key	Description	Required?
name	The name of the plugin	Yes
id	A unique identifier for the plugin, e.g. com.example.plugin	Yes
description	A description for the plugin	Yes
version	Numerical plugin version, needs to be increased to ensure agents are updated	Yes
default_n_historical	Integer representing how many previous values should be passed to classification logic. May be overridden from web interface.	Yes
default_classifier	File containing lua code to be used as the default classifier.	Yes
view	Either the name of an built-in view or "custom"	Yes
view_source	Name of the HTML file containing the view	If view is "custom"

Default Classifier This is a Lua file containing the code that will be used by default to classify the data from the plugin. This code will be loaded into the database on plugin installation and can be edited by users through the web interface.

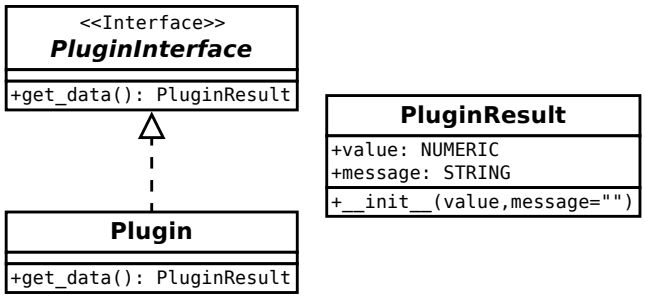
View Source Code If a "custom" view is specified in the manifest file, an HTML file must be included to be used as a view for the data. This HTML file is passed a list of collected data for the plugin for the date range specified for the report and is rendered using the Jinja2 templating engine. Any HTML or Javascript can be used to display the data as desired.

Python Code The plugin archive must be a Python module containing a class called `Plugin` which is a subclass of `PluginInterface`. Therefore, an `__init__.py` file must be included. There is no restriction to how the rest of the Python code in the plugin must be layed out however generally there will be a single Python source file containing the `Plugin` class which is exposed in `__init__.py`.

3.2.1 Plugin Logic

The plugin is a Python module and can therefore contain any logic that it needs in order to be able to retrieve the appropriate data. This may be as simple as using built in Python logic to get data such as CPU load, executing shell commands and parsing the result or reading system files. However, the external interface for a plugin must follow a defined structure. The structure of a plugin is shown in Figure 3.1.

Figure 3.1: UML diagram showing the structure of a plugin



All plugins must contain a class that extends from `PluginInterface`. This class must contain a `get_data()` method which is called by the agent to get the data from the plugin. The `get_data()` method must return and instance of `PluginResult`. The `PluginResult` object must contain a numerical value to represent the data collected by the plugin as well as a message which can contain more information about the data collected. The message defaults to being the empty string so does not need to be specified.

The plugin's main class (the one containing the `get_data()` method) must be exposed through `__init__.py` under the name `Plugin`. If the plugin's main class was called `CPULoad` in a module called `plugin.py` then it would be exposed using the line `from plugin import CPULoad as Plugin` in the `__init__.py` file.

Structuring plugins in this way provides a large amount of flexibility. The plugin can have any sort of internal implementation (it can even call out to shell scripts if desired) as long as the external interface follows the correct structure. Forcing plugins to return data in the form of the `PluginResult` object forces

them to return data in a sensible manner unlike existing tools which attempt to parse stdout of a shell script which is very fragile.

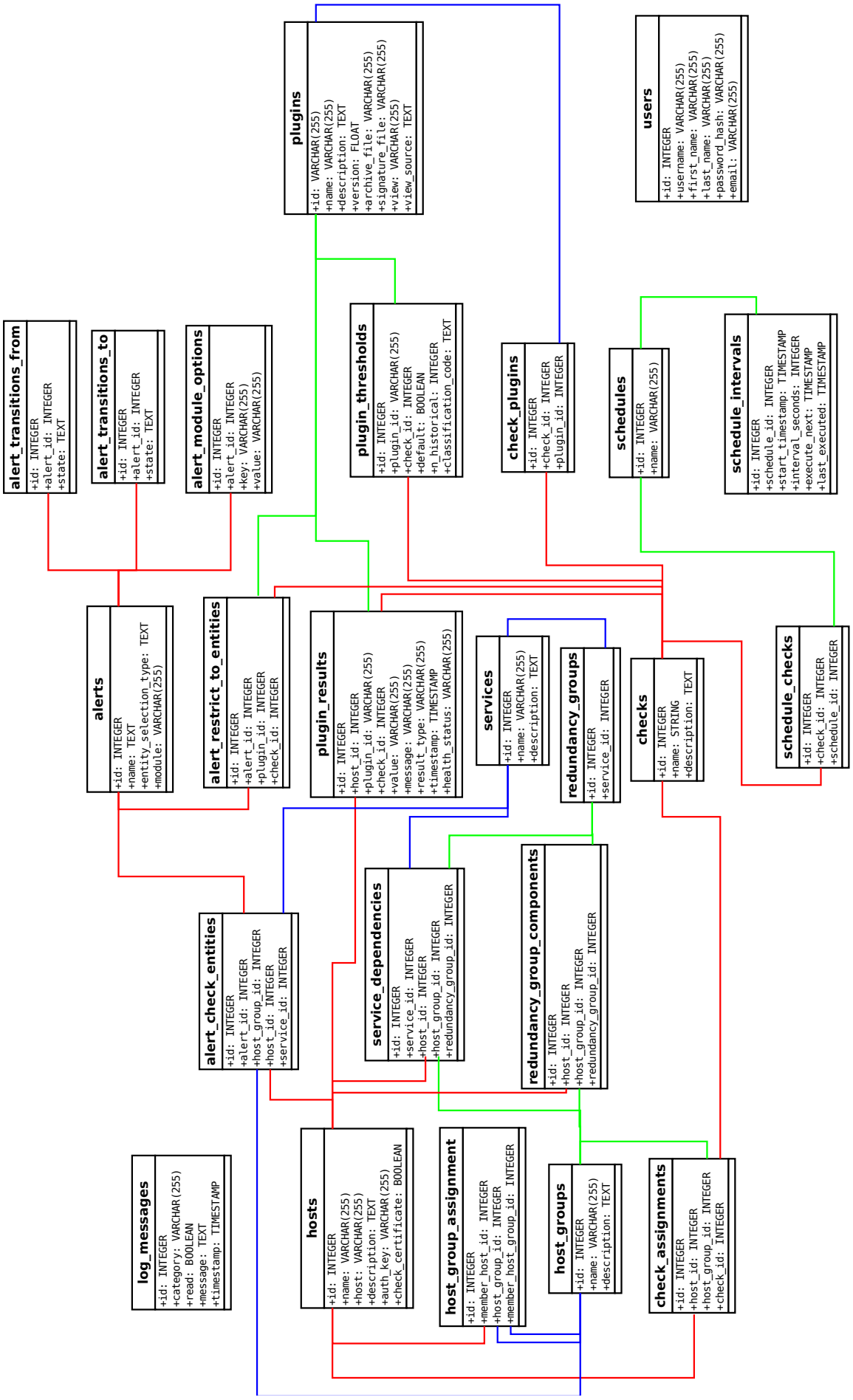
3.3 Database

The database is at the very center of Prophasis, it is used to store the configuration of the system along with data collected from agents, it is also the primary means of communication between the core and the web interface. Prophasis was designed to use the PostgreSQL RDBMS and great care was taken to ensure that the database design was built to be efficient and flexible enough to allow for future expansion.

The database was designed in a traditional manner using tables and relations between them in order to build an efficient and normalised design however it was implemented using the SQLAlchemy ORM which abstracts the database into a series of classes which saves time and reduces errors as queries do not need to be written manually. The manual design was carefully implemented using extensive use of SQLAlchemy's `relationship()` function. This allows relationships in the database to be traversed as easily as properties of an object. For example, if you have an entity `h` of class `Host` and an entity `r` of class `PluginResult` then it is possible to both access a list of all plugin results attributed to the host with `h.check_results` and to access, for example, the name of a host from the plugin result with `r.host.name`. The ORM handles all the logic of building the queries to gather the correct data. The ORM also makes it easy to edit data in the database, it is a simple case of retrieving the object, changing it's attributes and committing the session. It is also transactional which means that changes can be staged up and then either committed or rolled back, this is ideal as it makes it easy to abort from changes to the database in the event of an error. This also provides functionality to make multiple database operations atomic, this is important since there are multiple processes/threads writing to the same database.

Figure 3.2 shows a diagram that represents the structure of the entire database. The colours have no significance and are simply used to illustrate which lines are crossing over and which are not. This diagram was built in a tool called Dia and stored in an XML file which could be kept on version control, this meant that throughout the development process the diagram could be easily maintained to match changes to the database, in fact, the diagram was always updated before the changes were applied to the actual SQLAlchemy models. Having this diagram available was extremely useful when developing parts of the logic that utilise the database as it makes the layout much clearer than simply reading the source code for the models file or by looking at the actual, running database instance.

Figure 3.2: Diagram representing the Prophasis Database Schema



3.4 Common Module

Even though the core and web interface are completely separate processes, they still share some common logic: namely the database models, error logging and alert module handling. It would be impractical to write this functionality twice, especially the database models as this is a huge piece of work.

The best solution to this problem was to build a "common" python module known as `prophasis_common`. This contains the database models as well as error logging and alert module handling and is required by both the core and web interface. When installing either of these the common module must be installed first. Structuring the system this way prevents duplication of logic and ensures that there is no risk of there being any differences between the code in the core and the agent.

3.5 Core

3.5.1 Dispatcher

The dispatcher is the system which sends off requests to the agent and waits for the responses back which are then classified and then stored in the database. Due to the time it may take to execute some checks it would be impractical to execute them in series, therefore the dispatcher is multi-threaded.

The Prophasis dispatcher uses Python's built in "multiprocessing" library. This provides various methods to manage processes as well as thread-safe data structures for communicating between them. The dispatcher uses `multithreading.Process` to spawn worker processes and uses `multiprocessing.Queue` to define thread safe queues for passing data into these worker processes.

Fork vs Spawn The multiprocessing library supports three different ways to start a process. By default on UNIX systems it uses `os.fork()` to fork the existing process. This is unsuitable in this situation as the process creates a copy of the already open database connection - this causes conflicts when the workers try to write to the database. The solution to this is to tell the multiprocessing library to use the "spawn" context (the default on Windows) which will create a fresh Python interpreter process, this prevents open handles from being carried over into the child process. Using spawn is comparatively slower than forking the process but since we are only creating schedules are called (which is on a real world timescale), this difference will not be noticeable.

3.5.2 Classification

When data is collected from the agent it needs to be "classified" to determine whether it is "ok", "major", "critical".etc. Classification is done by Lua code that is bundled with the plugin and can also be modified by the user through the

web interface. The Lua classification code is stored in the database for ease of modification. The "lupa" Python library is used to integrate the Lua runtimes into the Python code.

3.5.2.1 Sandboxing

Classification code is executed directly on the machine under the same user as the core. Since this classification code can be changed through the Prophasis web interface it is critical that this code cannot perform malicious operations on the system. In order to resolve this, a Lua sandbox is created. This is done by creating a Lua table with only specific, trusted functions such as `math` and `ipairs` added to it. Lua's `setfenv` (set function environment) function is then called to ensure that all user provided code is executed inside this sandbox and can therefore not access more risky operating system functions such as file handling.

3.5.2.2 Functions

In order to make developing classification code easier, several predefined functions are provided to handle common operations such as `arrayMax(array)` which will return the maximum value in an array and `arrayContains(array, value)` which returns a boolean defining whether the given value is in the array or not. These functions are stored in a separate Lua file and are included before the user provided classification code before it is executed.

3.5.2.3 Handling Errors

When dealing with user defined code, there is always the potential for errors to occur when the classification code is executed. In this situation the system will automatically fall back and classify the result as "unknown". The error will also be logged within Prophasis and can be easily viewed by the user in the web interface's "System Logs" area.

- Scheduling
- Multi-threaded dispatcher

3.6 Agent

The agent is implemented using the Flask web framework to expose a HTTPS API that the core communicates with. Requests are sent to the agent using regular HTTP GET and POST requests with information passed using URL parameters or HTTP form data respectively. Responses are formatted as JSON. The Tornado HTTP server is used to handle incoming HTTP connections and communicates with Flask through using uWSGI.

Authentication In order to prevent unauthorised actions being performed on the agent, the core must authenticate with every request. The agent stores a hash of a long authentication key in its configuration file. This key is generated on agent installation and is different for every agent. This ensures that if the key for one agent is obtained, an attacker cannot access every other agent on the network. Storing a hash means that if someone was able to read the agent configuration file they cannot obtain the authentication key which could have allowed them to execute code as the agent's user which may have higher privileges than their user. HTTP's basic access authentication is used which allows a username and password to be easily sent along with an HTTP request, in this situation the username is "core" and the password is the authentication key. A Python decorator (`@requires_auth`) is applied to functions that require authentication which will verify the authentication token and only allow the function to execute if the token is correct, otherwise it will respond with HTTP error 401 (Unauthorised). Using a decorator keeps authentication logic out of the body of the function which ensures that the user is authenticated before the function is even entered. This prevents errors such as the authentication check being moved after some restricted functionality or accidentally being removed during changes to the function body.

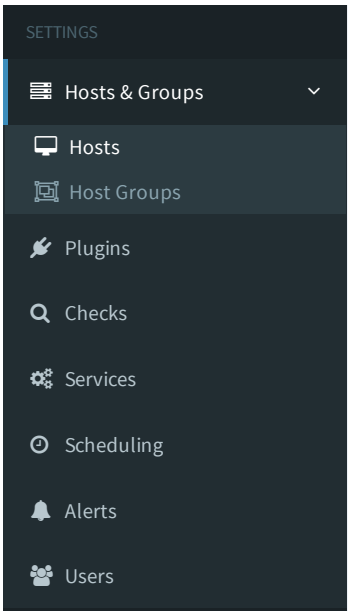
3.7 Web Interface

The web interface is implemented using the Flask web framework. The frontend is built using AdminLTE which in turn is based on Twitter's Bootstrap framework. The web interface is built to be fully responsive so that it performs equally as well on both desktop and mobile devices. The Jinja2 templating engine is used to render the pages from HTML templates. This allows certain pieces of template logic to be reused preventing code duplication and enforces a clear separation of logic between the processing and templating logic.

3.7.1 Configuration

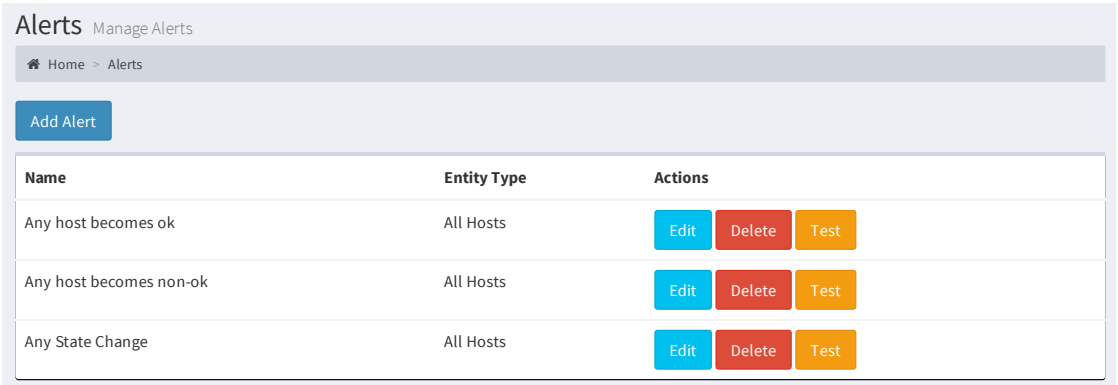
All of the different configuration options are accessible through a clearly organised menu displayed on every page. This allows the user to access all of the different configuration options from a single location. This navigation bar can be seen in Figure 3.3.

Figure 3.3: The "settings" navigation bar



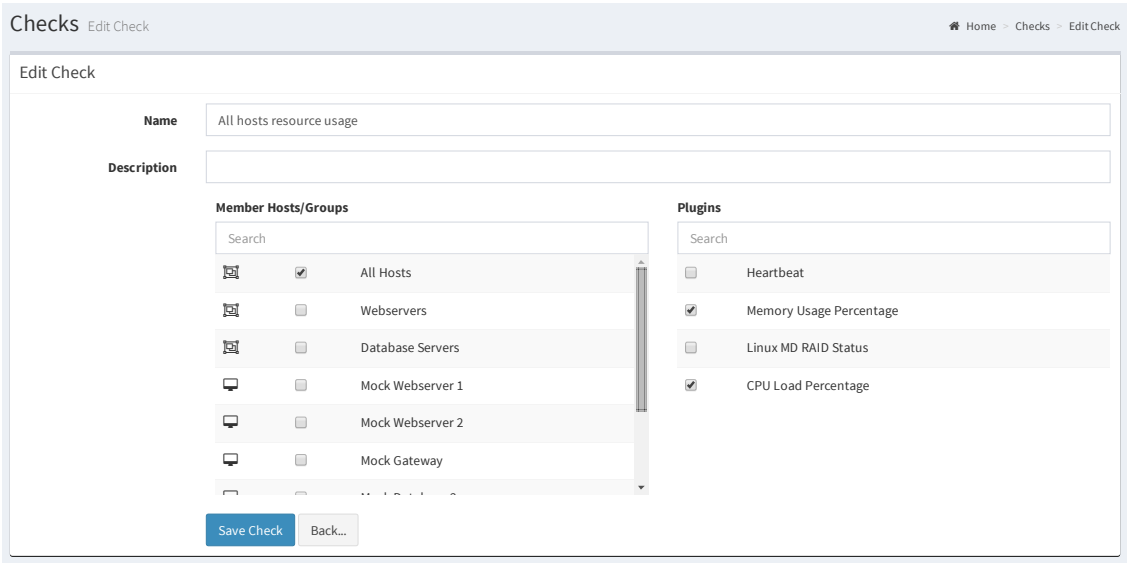
Clicking each of these links will take the user to the management area for that section allowing them to configure that part of the system. The first page of any of these sections is an index page that lists all objects (e.g. plugins or alerts) in the section with options to manage each of them as well as providing a link to add new objects. An example of the index page for alerts can be seen in Figure 3.4. This page lists all alerts in the system and provides buttons to edit, delete and test them. There is also a button allowing the user to add new alerts to the system.

Figure 3.4: The alerts index page



When the user chooses to add or edit an object a form is loaded. The add and edit forms are both rendered using the exact same template files ensuring that both forms are consistent. A `method` parameter is passed into these templates which is set to either "add" or "edit". This is used to adjust the template. Figure 3.5 shows the form for editing a check.

Figure 3.5: The Edit Check form

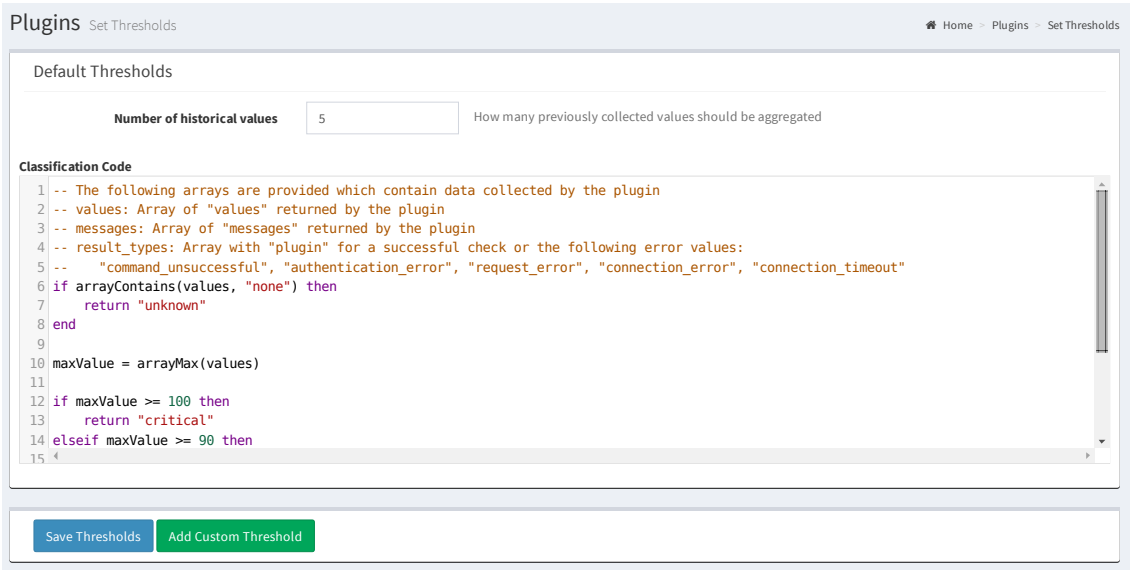


This form has a pair of text boxes to specify a name and description for the check and then a pair of lists of checkboxes: the first lists hosts and host groups that the check will be executed on and the second lists plugins that will be executed on those hosts when the check is run. The search boxes above each of these lists will filter the list to entries that contain the search term. This search system is provided entirely in Javascript and is generic enough to be applied to any HTML table by simply specifying some classes and HTML5 data attributes. These search boxes are applied to any tables in the system which may contain a lot of items.

Code editing As described in Section 2.2.2.3, Lua code is used to classify the results retrieved from plugins. In order to maintain the ability to manage everything through the web interface, it needs to be possible to edit this code through the web interface. For this, the CodeMirror[1] editor is used which provides an editor with syntax highlighting and support for using the tab key to indent blocks of code as well as intelligently handling indentation when pressing return. The form for editing classification code can be seen in Figure 3.6.

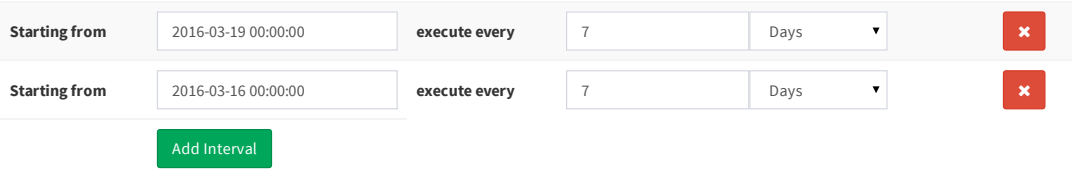
[1] https://codemirror.net/

Figure 3.6: The form for editing classification Lua code



Schedules As described in Section 2.3.4, a schedule is used to define when to execute one or more checks. The interface for this includes the usual form fields for a name for and description of the schedule as well as a checkbox list of available checks to execute. This page also has additional functionality to configure the time intervals at which the schedule is run. This can be seen in Figure 3.7. Each interval has fields to specify the start time for the schedule as well as a text box and drop down menu to enter the interval period. The green "Add Interval" button will add another row to this list allowing more intervals to be added and the red "X" button will remove the row it's in from the table. This is all handled purely in frontend Javascript.

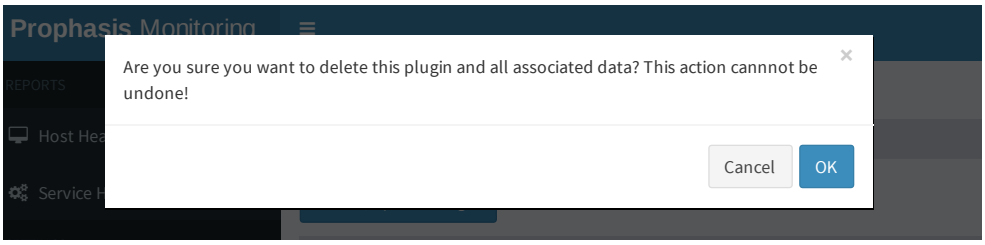
Figure 3.7: Form for managing the intervals for a schedule



In order to display confirmation messages to users, Bootbox.js[2] is used which provides a convenient abstraction around Bootstrap's build in modal functionality. This allows confirmation messages to be displayed to users without any manual HTML markup and responses to be collected straight into Javascript. An example of a confirmation message can be seen in Figure 3.8.

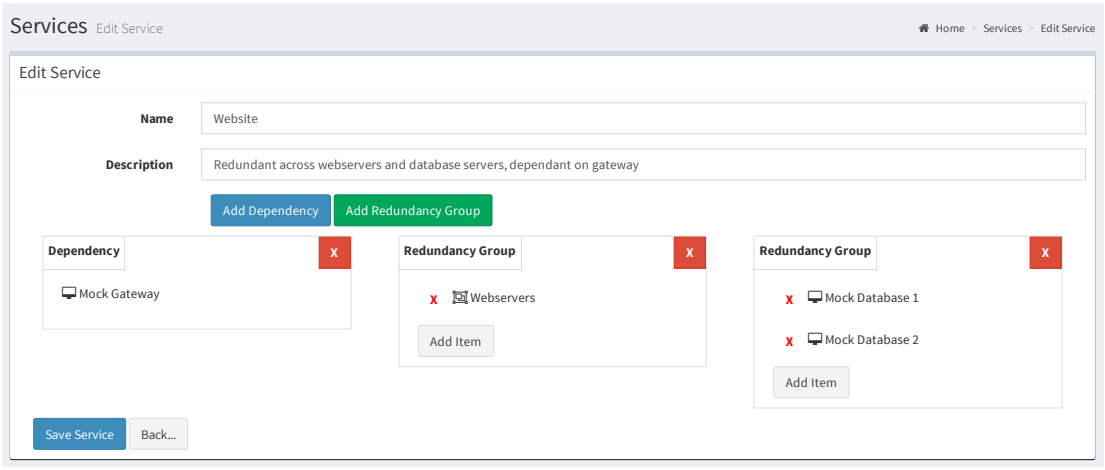
[2]http://bootboxjs.com/

Figure 3.8: The confirmation message displayed when deleting a plugin



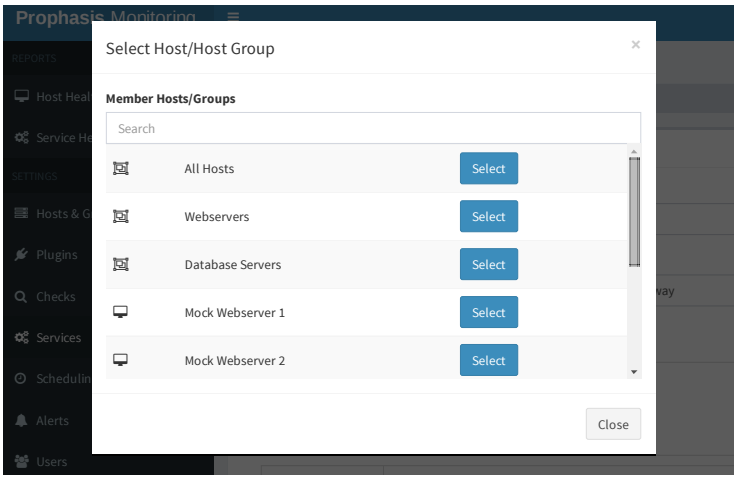
Services As described in Section 2.3.5, services are comprised of one or more dependencies or redundancy groups. Redundancy groups in turn are comprised of one or more hosts or host groups. Building a user interface to represent this was particularly challenging as data needed to be represented clearly and be intuitive to edit. The finished interface can be seen in Figure 3.9.

Figure 3.9: The user interface for editing services



In this interface, dependencies and redundancy groups are represented as boxes each containing the host(s) and host group(s) that are a member of that dependency or redundancy group. Individual hosts can be added to redundancy groups by clicking the "Add Item" button or removed by clicking the red "X" next to it. Entire dependencies can be removed by clicking the red "X" in the top right of the box. To select hosts or host groups to create a dependency or to add to a redundancy group, a Bootstrap modal dialog is displayed as shown in Figure 3.10. On this form, HTML5 data attributes are used to attach data such as IDs to the actual DOM elements that represent the structure of the service. When the form is saved, Javascript is used to serialise this structure for processing. This massively simplifies the code as adding and removing dependencies, redundancy groups, hosts.etc can be handled entirely in the DOM without having to also maintain a Javascript data structure at the same time.

Figure 3.10: The modal dialog for selecting hosts and host groups



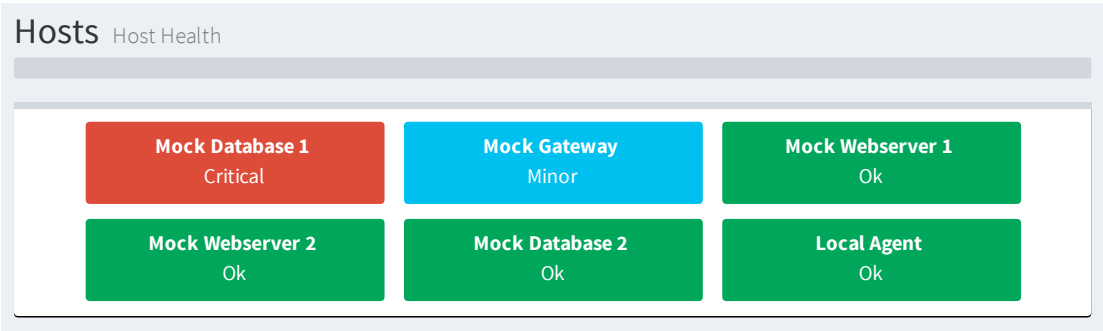
3.7.2 Reporting

In addition to being used to configure the system, the web interface is also used to visualise the collected data. There are currently two reports, one to show the health of all hosts in the system and another to show the health of all services. There is scope to add more reports to give different views of the data.

3.7.2.1 Host Health

Figure 3.11 shows the host health report for a selection of machines. This report lists all the machines in the system as well as displaying their health in both a textual and colour representation. The results here are sorted in order from least to most healthy. This ensures that even in a network with a large number of systems, unhealthy hosts will not go unnoticed due to being buried deep inside a list, especially on the small screens of mobile devices.

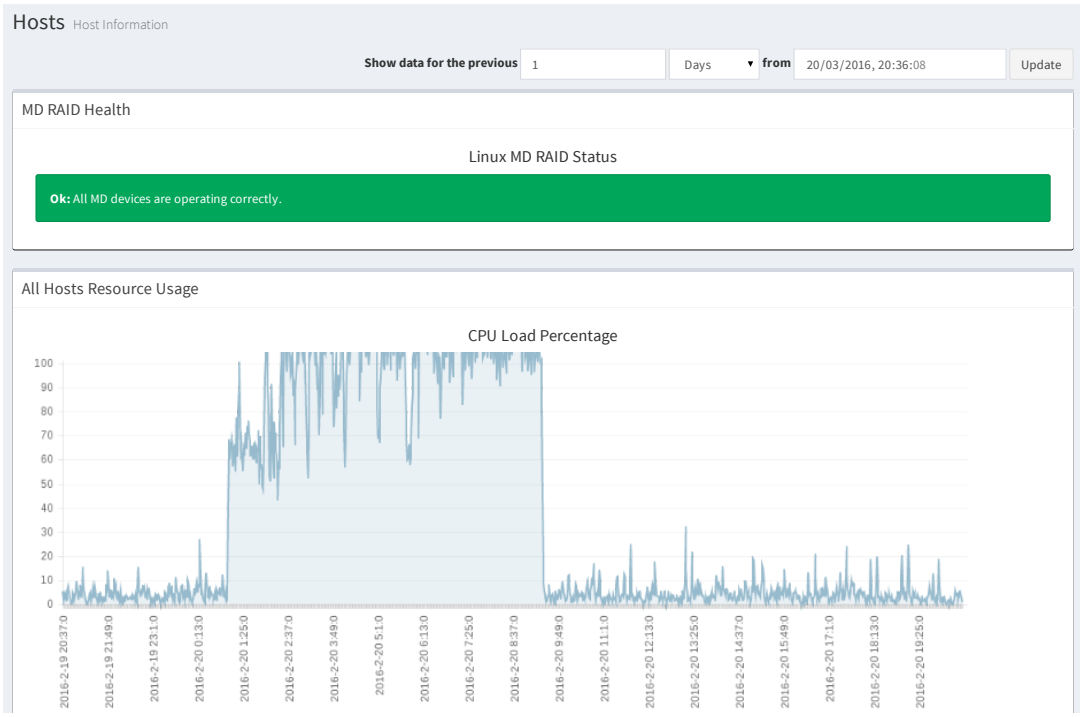
Figure 3.11: The index page for the Host Health report



Clicking on any of these hosts will open a page displaying more detailed breakdowns of the data stored for that host. This is where the timeseries data can be visualised. Figure 3.12 shows a sample output of timeseries data collected from a running server. This shows both the status of the Linux MD RAID devices as

well as a graph of CPU load showing a large spike due to a RAID consistency check that occurred on the machine. At the top of this report are a pair of inputs allowing the time period that the data represents to be specified.

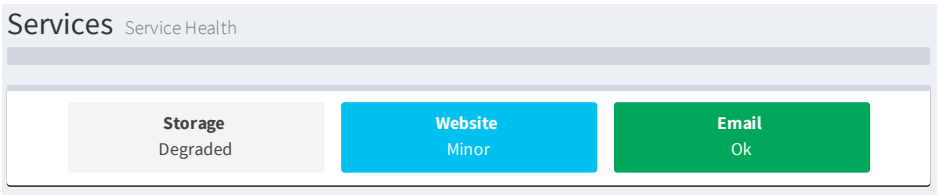
Figure 3.12: The host information page showing timeseries data for a single host



3.7.2.2 Service Health

In order to give a view of the overall health of the services running on a system or to show the impact that a fault with a host has on the overall functionality of a system, a report is available to list the health of all services. This is shown in Figure 3.13. Like the Host Health report, the services are ordered to show the services with the most serious condition first. As can be seen, services have an additional health status of "Degraded" which means that while a service is operating normally, one of its redundant components is experiencing an issue.

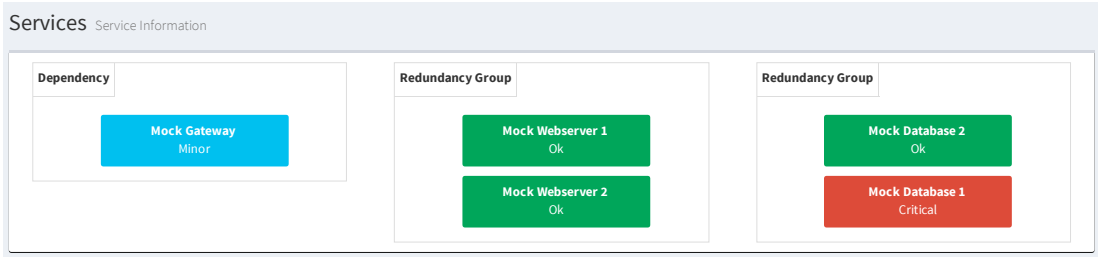
Figure 3.13: Page showing the overall health of all services in the system



Clicking on each of these services will load a view that shows the status of all hosts in the service as well as how they are structured in terms of dependencies and redundancy groups. This is shown in Figure 3.14. Clicking on each of the hosts on this page loads the host information view as shown above. This view

clearly shows the reason for a service's health status and allows issues affecting a service to be clearly seen.

Figure 3.14: Page showing the overall health of all services in the system



Chapter 4

Testing

4.1 Unit Testing

Unit tests are used to test the more complex functionality of the system such as getting all member hosts in a host group or determining the overall health of a host or service. These algorithms are somewhat complex so having unit tests is useful to both verify the functionality during development as well as to ensure this functionality is not broken by changes to other parts of the system. The Python `unittest` library is used as it included as part of a standard Python install and is well documented.

Each test case is implemented as a single class with each different test in a separate function within this. Each `TestCase` has a `setUp()` method which is called before each test in the class is executed. The `setUp()` method will drop all database tables, recreate the database structure from the models file and will then insert sample data for use in the tests. This ensures that every test starts on a clean database.

In order to prevent tests from affecting the main database or requiring a new database to be set up, SQLite is used. A temporary, in memory database is created when the tests are executed which is then destroyed when the tests are completed. When a test is started it sets an OS environment variable called `UNDER_TEST`. If this is set the models module will set the database connection string to `sqlite://` instead of the one stored in the config file. The SQLAlchemy ORM completely transparently handles the differences between PostgreSQL and SQLite therefore requiring no changes to the database code for use during testing.

4.2 Mock Agent

When developing and testing Prophasis, it is important to be able to simulate a network of multiple machines all being monitored and in addition to this it is useful to be able to simulate faults with these machines. While it would be

possible to run multiple agents and then use plugins to return data and to simulate faults this is impractical to set up, run and maintain.

As a solution to this, a mock agent was developed. This is a simple application that implements enough of the agent's functionality to allow the core to communicate with it as though it is a regular agent. It does not use plugins and instead reads the data to respond with from a JSON file stored on disk. The agent's `ping` method is implemented as normal, the `check_plugin_version` method always responds saying that an update for the plugin is not required (this means we don't need to implement the `update_plugin` method). The `get_plugin_data` method is then used to read the JSON file and respond with the appropriate data. Figure 4.1 shows an example of the JSON file used by the agent.

Figure 4.1: An example of the JSON file read by the mock agent

```
{
    "127.0.0.1:4048": {
        "me.camerongray.proj.cpu_load_percentage": {
            "value": 20, "message": ""},
        "me.camerongray.proj.memory_usage_percentage": {
            "value": 20, "message": ""}
    },
    "127.0.0.2:4048": {
        "me.camerongray.proj.cpu_load_percentage": {
            "value": 90, "message": ""},
        "me.camerongray.proj.memory_usage_percentage": {
            "value": 50, "message": ""}
    }
}
```

In order to keep the mock agent simple, it listens on a single port (4048, the same as the regular agent). The mock agent binds to all available IP addresses on the system and it uses the destination IP address to determine which data it is to respond with. In my usage I simply ran the mock agent on the same machine as the core and used loopback IP addresses picked from `127.0.0.0/8` to add multiple mock agents to the system giving each a different IP addresses. Each of these IP addresses is used as a key in the JSON file which maps to a JSON schema containing plugin IDs mapped to the data that should be sent in response to requests for that plugin.

The JSON file is read every time a request reaches the mock agent. This means that the test data can be changed on the fly while the mock agent is running by simply changing the data in the file and saving it. For example, to test what happens if the CPU load on a machine is too high one can simply change the value in the JSON file to the new CPU load and save it, the next time the CPU load plugin is executed on that specific agent, the new CPU load will then be sent out.

4.3 Use in the Field

There is only so much information that can be gained from testing a system such as Prophasis in a clean environment through the use of the mock agent and unit tests. In order to see how Prophasis performs when deployed in a real world environment I installed and ran Prophasis in two different real world environments.

4.3.1 The Tardis Project

The Tardis Project[1] is a computer system operated and maintained by students at the University of Edinburgh. It provides services to students including shell accounts, web hosting and mail services. The project currently runs off of 9 physical servers and on these operates a large number of virtual machines.

Prophasis was installed on the Tardis system somewhat early in its development so that it could be operated in a running environment 24 hours a day. A virtual machine was created to run the core, web interface, an agent and the database, the agent was also installed on several different hosts including the shell server and the machine that provides network storage to the rest of the system.

Tardis's primary storage host uses Linux MD RAID to provide redundancy. For this a custom plugin was developed to parse the file at `/proc/mdstat` which provides information about the health of the disks in the system. This allowed Prophasis to be able to monitor the health of the Tardis system's primary storage.

The deployment on Tardis was extremely useful as it allowed Prophasis to be tested in a real world environment from the very beginning where new features could be launched and tested on live servers rapidly. It also highlighted issues that would otherwise be hard to find such as those that only occurred infrequently or after the system had been running for a reasonable period of time.

Scheduler Deadlock One issue that was discovered while running on Tardis is where the scheduler could, in rare cases completely lock preventing future checks from being executed. This issue was caused where a check could take too long to run resulting in the time at which it was next meant to execute being set to a time which, by the time the check had completed, was in the past. Therefore the check would never be executed again. This issue was later solved by allowing checks to execute "late" up until a defined threshold, if they are later than this threshold, the time at which the check is to execute next will still be incremented. This prevents this locking issue from occurring. Since this issue occurred very rarely, it was extremely useful to be able to use Tardis as the core could be left running 24/7 which allowed this issue to be noticed, if Prophasis had only ever been run on a single machine for short periods of time during development, it is possible that this issue would never have been discovered.

[1] http://www.tardis.ed.ac.uk/

4.3.2 Lynchpin Analytics Limited

My employer, Lynchpin Analytics Limited is full service analytics consultancy. Lynchpin is a small business without a dedicated systems administrator yet operates a reasonably sized network running entirely on their own hardware. This is the sort of environment which Prophasis is aimed at and where existing tools fall short. Prophasis was deployed at Lynchpin when it was nearly complete in order to evaluate how the finished product works in a heterogeneous environment and to highlight any major issues that occur.

Lynchpin operates a selection of vastly different systems which makes it an ideal environment to test Prophasis. Unlike on Tardis where the majority of systems run Debian, Lynchpin operates systems running Citrix XenServer (where Dom0 is based on CentOS 5), Debian and FreeBSD with a wide variety of different types of hardware. The Prophsis core, database and web interface were installed in a Debian Linux Virtual Machine running on one of the XenServer hosts. The agent was then installed on several different systems. Table 4.1 lists the systems on which the Prophasis agent was installed. This allowed me to test Prophasis on a selection of different operating system families (Linux, BSD and Windows) and storage configurations (Linux MD RAID, Dell Hardware RAID and ZFS on top of an HP SmartArray controller).

Table 4.1: Systems on which Prophasis was installed

Type	**Operating System**	**Storage**
Virtual Machine	Debian 8 (Jessie)	Single Virtual Disk Image
Virtual Machine	Windows Server 2012 R2	Single Virtual Disk Image
HP ProLiant DL180 g6	FreeBSD 10.2	ZFS, HP SmartArray P410
HP ProLiant DL180 g6	FreeBSD 10.2	ZFS, HP SmartArray P410
Dell CS24-SC	Citrix XenServer 6.5	Linux MD (Software) RAID
Dell CS24-SC	Citrix XenServer 6.5	Linux MD (Software) RAID
Dell PowerEdge R630	Citrix XenServer 6.5	Dell PERC H330 RAID
Dell PowerEdge R630	Citrix XenServer 6.5	Dell PERC H330 RAID

The deployment was extremely successful with Prophasis reliably monitoring all systems. Performing this deployment on Lynchpin's systems was extremely valuable and both demonstrated that Prophasis works well in such a deployment and highlighted some improvements that could be made.

Stability The deployment of Prophasis at Lynchpin has demonstrated the stability of Prophasis running in a live environment. The entire system operated successfully for more than a month without requiring any component of the system to be rebooted. Over this time a large amount of historical data has been collected and stored within the Prophasis database without causing any sort of major performance degradation, this shows that the system will not slow down

over time due to the database filling up. There was also no incidents of the scheduler locking which, while it doesn't prove the absence of issues of this type, shows that the scheduler is stable enough for production use.

Documentation Deploying Prophasis on several different systems at Lynchpin gave a valuable opportunity to document the install process on several different platforms. This allowed installation instructions to be written for Debian Linux, Citrix XenServer, FreeBSD and Windows. This also gave insight into subtle differences in the availability of different dependencies on different platforms such as the lack of Python 3 in the XenServer (CentOS 5) repositories, requiring it to be built from source.

Plugin Development The large number of different storage configurations on which Prophasis was deployed provided a valuable opportunity to develop plugins for the different storage configurations. A plugin was developed to read the health of the drives running on an HP SmartArray P410 RAID controller under FreeBSD through the use of `camcontrol`. Another plugin was written to check the health of a ZFS zpool which executes `zpool status` and parses the output. In order to check the health of the disks running behind the Dell PERC H330 RAID controller, a plugin was developed which read SMART data by calling `smartctl` with `-d megaraid,nn` to access the disks behind the Dell PERC (a rebranded LSI MegaRAID) RAID controller.

Plugin Execution Timeouts While testing the deployment at Lynchpin, an issue was discovered with the plugin that was checking the zpool status on the machines using ZFS. One of the hosts developed an issue resulting in the `zpool status` command simply hanging. Prophasis had no timeout for this so once this plugin had been executed, any future plugins would be stuck in the dispatch queue in the core and would never be executed (due to the one simultaneous check per host policy). The solution for this was to implement timeouts on the agent so that if a plugin takes longer than a predetermined time limit to execute, the execution will be terminated and an error would be logged. This prevented the dispatch queue from being blocked up in the event of a long/infinitely running plugin.

Windows Support The agent was deployed on a virtual machine running Windows Server 2012 R2 Standard, this provided an opportunity to test the behaviour of the Prophasis agent when running under Windows (previously it had only ever been executed on UNIX-like operating systems). It was found that the "timeout-decorator" library being used to allow plugin execution to halted if a plugin was taking too long to execute used `SIGALARM` signals which are not supported by Windows.

Ideas for Future Work My experiences of deploying and using Prophasis at Lynchpin also gave me several ideas for future features that could be implemented. While the system was operating, one of the XenServer hosts decided to check the

consistency of its RAID arrays. This resulted in extremely high CPU load which sent out alerts warning of this issue despite this being standard operation. Future work to solve this could involve allowing data from one plugin to be retrieved when classifying data from another plugin. This would mean that the CPU load classification logic could be configured to ignore CPU load caused by MD RAID checks on any hosts which use Linux MD RAID. It was also determined that it would be useful to be able to pass arguments into plugins from the core. This means that, for example, a plugin could be written to check the free space available on a disk and the disk to check could be specified through an argument to the plugin. This would allow different checks to be created to check the free space on different disks. These features are described in more detail in Section 5.1

Testing Prophasis by deploying it at Lynchpin was incredibly useful. Lynchpin as a company perfectly suits the type of environment that Prophasis is designed for, this shows that Prophasis works reliably in such an environment. It also provided a valuable opportunity to test on a wide selection of software platforms and hardware configurations to ensure software support. Deployment also indicated some potential issues which have now been resolved as well as inspiring ideas for future features and enhancements.

Chapter 5

Evaluation

Prophasis is now a complete system that is sufficiently complete that it could be deployed on a network. It has been tested both in a synthetic environment as well as on multiple real world networks. The complete system improves on the points at which current systems tend to fall short as described in Section 1.2, these enhancements are described below.

Time series monitoring and real time alerting Prophasis has support for both time series monitoring and real time alerting built in. Time series data is tightly integrated into the system rather than being bolted on later meaning that historical data can be used to make decisions such as the overall health of a host. This is a clear improvement on existing tools where they traditionally only support time series monitoring or real time alerting.

Support for Custom Metrics Like all other tools, Prophasis supports monitoring custom metrics using custom code. However, Prophasis has a much more rigid structure for plugins which helps to ensure consistency and reliability of plugins.

Classification Threshold Definitions This is an area where Prophasis clearly improves on existing systems. Other systems rely on the plugin code that is executed on the remote host to determine and return the overall classification which means that classification logic is stored all over the place. In Prophasis, plugins simply return a raw value that represents the collected data, classification is then done by user provided Lua code on the monitoring server itself. This means that classification code is all stored in one place and can be edited directly from the web interface. Classification code also has access to historical data so it can make better decisions than it could if it only knew the current state of a host. Using short blocks of Lua code provides total flexibility to allow any sort of logic to be used to classify the data instead of simply having rigid thresholds defined through a form in the web interface.

Code/Configuration Delivery to Nodes This is another clear area for improvement. Current systems rely on code for custom plugins to be stored on the

machines being monitored but do not provide functionality to handle this. Therefore a separate tool such as Puppet or Ansible would be required to manage this, tools such as this may not be in place on smaller networks where each machine has a very distinct role which reduces the need for centralised configuration management. On the other hand, Prophasis stores a central repository of all plugins on the monitoring server and distributes these to remote agents automatically. This means that plugins only need to be updated in one place by uploading the plugin archive through the web interface, Prophasis transparently handles the rest.

How the user configures the system Current tools also store all of their configuration settings in files stored on disk, these require a deep understanding of the system to configure and maintain. In Prophasis, configuration files only store the minimum amount of information required for the system to be able to start such as database connection settings. The majority of configuration relating to checks, alerts, schedules.etc is stored in the database and managed through clearly laid out forms in the web interface. The web interface contains help text and explanations of various terms as well as validating user provided input to prevent invalid configuration being saved.

How Dependencies are Handled Prophasis has a flexible system for handling dependencies. A service can be configured as being dependent on all specified hosts being accessible (an `AND` dependency) or can be configured for redundant systems where the availability of a service requires at least one host from a set of hosts to be accessible (an `OR` dependency). Various different combinations of `AND` and `OR` dependencies are possible. This allows alerts to be fine tuned preventing unnecessary alerts resulting from minor faults in a highly redundant environment.

5.1 Future Work

While Prophasis is now a complete system which could be (and has been) used in production environment in its current form, there is considerable additional functionality and enhancements which could be added to the system to improve it even further.

Distributed Monitoring Currently, the Prophasis core runs on a single host and monitors all other hosts in the network. This is sufficient for most small to medium sized networks (the ones Prophasis targets) but provides no redundancy in terms of the monitoring system. If the Prophasis host were to fail then no alerts would be sent out. This could be made worse if Prophasis shared a physical machine with more critical services and this machine was to fail as critical services would become unavailable but since Prophasis has also gone down, nobody would be notified. A solution to this would be to build in support for multiple cores to be operating simultaneously where each monitors it's own set of nodes. This could be implemented through the use of database replication to keep the database in sync

across all monitoring nodes. One node would then be designated the "master" and would handle the web interface, all other nodes can only insert the results of checks into the database which would reduce the complexity involved. This could be enhanced with functionality to automatically "repair" the monitoring system in the event of a monitoring node failure by assigning the hosts that the failed node was responsible for monitoring amongst the remaining monitoring nodes.

Server-side Plugins Currently, plugins are executed exclusively on the host they are checking. This additional feature would allow plugins to be marked as "server-side" meaning that they are executed by the core and run on the monitoring node instead of the agent. Plugins of this type could check external availability of services running on a host such as ensuring that a web server is listening externally and that no firewalls are blocking traffic. Server-side plugins could also be used to check devices that cannot run the Agent directly such as routers and switches. In this case a server-side plugin could be written to communicate with these devices over SNMP.

Access another plugin's data in classification logic Currently, the classification logic for a plugin can only obtain the previous n values that were collected for that plugin itself. This feature would allow one plugin to read data from another plugin for use when it is deciding how to classify the data. For example, a CPU load monitoring plugin may want to pull in a list of running processes so that it does not alert about high CPU load when a known "safe" process is using a large amount of CPU. Another example of this could be for a plugin that monitors the temperatures in a system where it may want to have different temperature thresholds based on the current CPU load of the system.

Passing arguments into plugins This feature would allow plugins to accept arguments when they are executed. These arguments would be specified at the check level so that multiple checks can be configured to execute the same plugin but with different arguments. An example of this would be a plugin to check if a certain process is running on a system. It would be impractical to have a different plugin to check each process so a general one could be developed where the name of the process it is checking for can be passed in as an argument. It would then be possible to have a check to check if the web server is running by passing "httpd" in as the argument and another check that uses the same plugin to check that the database server is running where "postgres" is passed in as the argument.

External API An API could be developed to allow data collected by Prophasis to be accessed. This would allow external tools to be integrated with for various different tools. This could allow deeper analysis and better use of the collected data such as automatically provisioning additional servers on a cloud platform in the event of current hosts becoming overloaded. This API could also be used to manage the configuration of the system such as adding new hosts automatically

which would be particularly important in cloud environments where hosts are created and destroyed on a regular basis, often automatically.

Documentation Currently there is very little in the way of documentation explaining how to configure and operate the system as well as information about developing plugins. A large amount of future work would be to write comprehensive documentation about the system instead of relying on using the source code as a reference.

Nested Services Currently services can be defined as a set of `AND` and `OR` dependencies across hosts and host groups. This functionality could be enhanced by allowing these dependencies to factor in the health of other services thereby reducing duplication of dependency structure. For example, a website may be redundant across multiple web server hosts and also be dependant on the availability of the storage service.

Chapter 6

Conclusion