UNIVERSITY OF ST ANDREWS

SENIOR HONOURS PROJECT

OmniPanel: A Web Control Panel For Distributed, Heterogeneous Applications

David Jones

Supervisor:
Dr Adam Barker

Second marker:
Professor Saleem Bhatti

April 13, 2012

Abstract

The aim of this project is to develop an extensible middleware capable of exposing the functionality of many distributed command-line applications within a single web control panel.
With the rise of Cloud Computing and Infrastructure As A Service, it is becoming increasingly common for web services owned by a single entity to be provisioned within separate virtualised containers. As a result of this movement, the mechanisms that facilitate interaction with these web services are also becoming separated, to the extent where significant time is spent managing and swapping between different administrative tools.
OmniPanel is a framework that allows users to build a personalised web interface that amalgamates the functionality of their remote applications within a single web-page.

The system has been developed with scalability in mind such that many remote applications can be manipulated within a single session. Furthermore, the library of remote applications that OmniPanel supports can easily be augmented by third-party developers thanks to the standardised manner in which OmniPanel exposes its functionality and its data.
OmniPanel has been used successfully to interact with a simple Java application deployed upon multiple remote hosts and for the monitoring and administration of a popular gaming server. Early performance testing suggests that OmniPanel will scale gracefully to meet the needs of many simultaneous users.

Declaration

Contents

Chapter 1

Introduction

1.1 Problem Specification

The aim of this Senior Honours project is to develop a web-framework that allows for the simple administration of multiple server instances, all from a single page. The user can build up a personalised control page by deploying multiple *widgets* within free *slots* on the interface. Widgets are configured on deployment to include the IP address(es) of the remote machine to be controlled along with login credentials required to open SSH connection(s).
The motivation for this project is to do away with the need for multiple browser tabs, all containing a web control panel for a single host or service. Now everything can be monitored and manipulated from a single place which, once configured, can be accessed from any web browser in the world. Furthermore, the framework has been developed in such a way that new functionality can easily be added by external developers who must simply write the code for the html front-end and python back-end files. This makes the project hugely flexible allowing the functionality of any command-line application to be exposed within a web interface. Crucially, the complex routing of data both internal and networked (via SSH) remains hidden from the user at all times, creating a transparent routing middleware. This middleware is named OmniPanel.

1.2 Useful Definitions

1.2.1 Widget

A widget is a section of HTML that can be placed within a slot on a user's control page and with which the user can interact. Widgets are defined as HTML containing special place-holders that are later overwritten by OmniPanel with real values (showing the current state of a remote application). Widgets also contain HTML form elements that when submitted, are processed by OmniPanel and an associated back-end script to manipulate the state of a remote application.

1.2.2 Deployment

A deployment is an instantiation of widget: The widget has been placed on the user's control page and the user has provided the IP address(es) and login credentials needed to allow the widget to monitor and manipulate a remote host.

1.2.3 Back-end

A back-end is a python script that is directly associated with a widget. When a user interacts with a widget, any values they submit are passed to the associated back-end script. The back-end is responsible for listing commands that need to be executed on a remote server and also parsing a server's response to certain commands so that remote applcation state can be determined.

1.2.4 OmniPanel

OmniPanel can be considered as the glue that joins front-end widgets and back-end scripts together. It is responsible for the correct routing of data, the provision of a user's control panel and the issueing of commands over SSH to a remote host.

1.2.5 Host

A host is a remote computer, specified by the user when deploying a widget and with which OmniPanel interacts via SSH. It is the machine that we contact to ask for an update of state and the machine we send commands to in order to interact with an application upon it.

1.2.6 Screen

Screen [1] is an extremely useful tool that runs under the linux and Mac OS operating systems. It allows for a single user session to be multiplexed over multiple connections such that many different users can all interact with the same instance of a program, running within a screen. Normally, unix based operating systems isolate user sessions such that the running of a program in one session is totally invisible to anyone else using that same system. Furthermore, when that user ends their session (by tearing down their SSH connection, for example), the application they are running will be torn-down too. Screen allows you to 'detach' from the application prior to disconnect such that the program continues to run, even when the user has left. Some applications, such as an apache server, run as a service daemon on a host and are therefore persistant and visible across all sessions. For applications that do not run as a service, such as a Counter-Strike gaming server (simply an executable binary), screen sessions are invaluable. For this reason, my application fully supports interactions with applications that are running within screens on a remote host.

[1] http://linux.die.net/man/1/screen

1.3 Project success

A complete, functional implementation of the system has been achieved that meets all the primary requirements outlined in the project proposal. The OmniPanel sytem appears robust under trial and is written at a sufficiently high level such that new functionality can be easily added by external developers. OmniPanel facilitates the monitoring and manipulation of one or more remote applications through user interaction with a web interface.
Care has been taken to implement the system in a scalable manner, such that many servers can be monitored and interacted with in parallel, with components being carefully managed to preserve system resources. Initial performance data suggests that OmniPanel does indeed scale well when interacting with multiple remote hosts.

1.4 Outline of report

Objectives: Specification of the primary, secondary and tertiary project objectives.
Requirements: Specification and description of the functional and non-functional requirements for each part of system (Web Interface, OmniPanel and Back-End Script).
Context Survey: Discussion of technological fields related to the OmniPanel project and existing systems that solve similar problems.
Ethics: A discussion of the ethical issues related to the OmniPanel project.
System Architecture: Discussion of OmniPanels architecture showing the interaction of components to achieve project objectives.
User Manual: An introduction to OmniPanels user interface explaining how it should be used.
Software Engineering Techniques & Processes: A discussion of the various engineering techniques employed throughout OmniPanels development.
Implementation: A discussion of the various technologies leveraged to implement OmniPanel.
Testing: A description of the testing methodology employed for OmniPanel presenting the results of both function and scalability testing.
Evaluation: Provides a discussion of testing results and compares OmniPanel against original specification and existing systems. Concludes with future work for the project.
Conclusion: Highlights the main achievements of the project and provides a final conclusion on project success.

Chapter 2

Objectives

2.1 Primary Objectives

- Allow each user to create their own personalised control page with deployed widgets remaining persistant between sessions.
- Allow user to monitor and interact with a remote server using a deployed widget.
- Allow user to extend the functionality of the system by uploading new front-end widget and back-end scripts that can then be deployed.

2.2 Secondary Objectives

- Allow for a single widget deployment to monitor and control multiple remote hosts.
- Develop a range of front-end and back-end files to fully demonstrate the flexibility of the system.

2.3 Tertiary Objectives

- Provide facility for modifying an existing deployment, for example, adding or removing a host to be monitored and controlled.

Chapter 3

Requirements

3.1 Control Page

1. Users can deploy widgets upon their control page to form a configuration that persists between sessions.	
Type	Functional
Priority	1
Description	The system must be able to store a user′s control page configuration against their user account so that it remains the same across sessions. Deployments must retain host information (IPs and user credentials) along with their position on screen (which slot they are placed in).

Users can issue commands to a remote host through interactions with a widget on their control page.	
Type	Functional
Priority	1
Description	User interactions must be communicated effectively with the back-end script such that suitable commands can be generated and sent to the server (via OmniPanel) to fulfill the user′s request.

2. Widgets on screen must automatically update with new content.	
Type	Functional
Priority	1
Description	It is important that widgets automatically update themselves to show a new state, without requiring interaction from the user. Furthermore, it should not be necessary to reload the entire page for each update as this will deduct significantly from the user experience.

User can specify an unlimited number of hosts to be associated with a single widget.	
Type	Functional
Priority	2
Description	Front-end must be designed in such a way that an unlimited number of IP addresses can be entered by the user.

On a multi-host deployment, the user should be able to select which host they want to monitor at any given time.	
Type	Functional
Priority	2
Description	Having a single widget configured against multiple remote hosts works perfectly for issueing commands but not for monitoring state (there is no guarantee all hosts share exactly the same state). The user must therefore be able to dynamically change which individual host they wish to monitor.

The user should be able to adjust the frequency at which a widget polls for an update.	
Type	Functional
Priority	2
Description	The frequency with which a new state needs to be retrieved from a host depends on the application running upon it. Monitoring a game server will likely require far more regular updates than the monitoring of a file-system, for example.

It should be possible for hosts to be automatically discovered on a network without the need for manual user input.	
Type	Functional
Priority	2
Description	In situations where many hosts need to be controlled via one widget, it would be desirable to avoid manual IP input. A host discovery mechanism based on UDP multi-cast should help us to automatically discover clients on the local network.

It must be possible for a user to extend the functionality of the system to meet their needs.	
Type	Functional
Priority	1
Description	The system must be developed in such a way that the middleware is abstract enough for any front-end and back-end implementations to integrate successfully. The user could upload custom front-end and back-end files via the HTML control page.

User interactions must be realised within an acceptable amount of time.	
Type	Non-Functional
Priority	1
Description	When a user interacts with a widget to issue a command, they should see the result of this command soon after. Failure to meet this requirement will likely result in repeat interactions thus leading to unwanted state on the remote host.

Users must be made aware of any errors that occur within the system.	
Type	Functional
Priority	1
Description	If an error occurs within the system, for example, an SSH connection cannot be established due to incorrect credentials, the user must be made aware of this.

The system must respond gracefully to unexpected events.	
Type	Non-Functional
Priority	1
Description	Since the system integrates directly with remote systems outside of our administration, it must be able to tolerate external misbehaviour e.g. a host crashing or abruptly terminating an SSH connection. Automatic reconnection should be attempted and the user should be informed.

3.2 OmniPanel Core

Middleware must correctly route user interactions to a back-end script for processing.	
Type	Functional
Priority	1
Description	It is essential that the correct interactions are routed to the correct back-end scripts else there is a real danger of nonsense commands being transmitted to the server(s).

Middleware must expose data in a predictable, consistent manner.	
Type	Functional
Priority	1
Description	In order for external developers to write custom front-end and back-end files, the middle-ware must behave in a predictable manner. The same data structures must always be passed between middleware and back-end and these structures must contain the minimum amount of data needed for the back-end to function correctly.

Middleware must be able to manage SSH connections for transmission of commands on behalf of the user.	
Type	Functional
Priority	1
Description	SSH connections must be created on demand, stored and torn-down as required by the user.

The OmniPanel core should be able to interact with a remote host without the need to modify the host.	
Type	Functional
Priority	1
Description	Requiring that a remote host be modified in some way (installing software, opening ports, adding certificates) will reduce uptake of the OmniPanel system.

System must promote scaleability by allowing the reclaim of resources	
Type	Functional
Priority	1
Description	As the user interacts with the system, the resources they require (open SSH connections, communication with back-end scripts) will vary. The system must intelligently track which resources are no longer needed and reclaim them for use by other users.

System must promote scaleability by allowing the sharing of resources	
Type	Functional
Priority	1
Description	The system must be able to detect scenarios where existing resources can be shared by multiple components. For example, when two deployments monitor the same screen session on the same remote host, there is no need to setup two connections to poll for state. One poll can support both front-end updates.

System must promote scaleability through the synchronisation of client $\rightarrow$ middleware, middleware $\rightarrow$ server polling	
Type	Functional
Priority	2
Description	If the user has set their front-end widget to poll for an update every 20 seconds then there is no need for the middle-ware to be contacting the server every 5 seconds - The majority of the updates will never be seen by the user. The middleware should match it's rate of polling to that of the client.

System must ensure sensitive user data is kept secure at all times	
Type	Functional
Priority	1
Description	Files containing sensitive data should be encrypted using the latest in cryptographic functions. Furthermore, these files should be automatically deleted from the system when they are no longer needed.

The system should promote third-party development.	
Type	Non-Functional
Priority	1
Description	Since the success of OmniPanel is highly dependant on it's ability to interact with a range of remote applications, we must encourage third-party development by exposing functionality and data in a standardised, well-understood manner.

The system should be easily deployed upon a standard unix web-server.	
Type	Non-Functional
Priority	1
Description	To ensure that OmniPanel meets the demands of its user-base, we must ensure that it is easily deployed on a standard unix web-server at short notice. This way, traffic load can be balanced across multiple servers to cope with times of high-demand.

3.3 Back-end Scripts

Back-end must be able to process a user's interaction	
Type	Functional
Priority	1
Description	Submitted data from the user's interaction should be passed to the back-end script which can then output the commands needed to realise the user's request

Back-end must be able to supply instructions needed to query server state	
Type	Functional
Priority	1
Description	The commands needed to retrieve server state will vary between application and therefore will need to be passed from the back-end to the middleware

Back-end must be able to process a server's response	
Type	Functional
Priority	1
Description	Once the commands have been sent to query a server state, the response must be parsed to extract the relevant information.

Back-end must be able to supply the middleware with new HTML placeholder values to be rendered in widget.	
Type	Functional
Priority	1
Description	After extracting relevant information from server responses, the back-end must be able to match this information to HTML placeholders so it can be displayed to the user in their widget.

Chapter 4

Context Survey

4.1 Key Areas

4.1.1 Cloud Computing

Cloud computing refers to applications that are delivered over the internet and the range of technologies that underpin their functionality [1].
Closely coupled to the emergence of cloud computing is the development of Infrastructure As A Service (IAAS); a system in which IT infrastructure is delivered by virtual machines housed within large data centres [9].
Cloud computing is becoming ever popular with revenue predicted to grow from $56B in 2009 to $150B in 2013 [10]. One of the primary drivers for this mass uptake is a companys ability to migrate their computer systems to external providers, thus outsourcing the responsibility of maintenance and furthermore, only paying for the computational resources they use. [1]
With companies such as Amazon's Elastic Computer Cloud (EC2) making cloud computing increasingly accessible to the masses, it is common for systems that once stood within the same physical location to now be spread between different data centres located all over the globe. This is a primary motivator for OmniPanel's development, making it possible for system administrators to amalgamate geographically distributed systems in to a single control page through which they can monitor and manipulate the entirety of their system.

4.1.2 Cluster Administration

Since the 1980's, computer architects have been trying to find ways to create increasingly powerful and increasingly efficient computer processors. This began with attempts to improve the traditional, sequential computer processor before deviating in to the realms of expensive parallel computing. In the early 90's, scientists began to investigate the use of many cheap, commodity computers, the resources of which could be combined to help solve a single, complex problem [2]. This collaboration of many low-specification devices is known commonly as Cluster Computing.
Navarro [6] describes how the emergence of Cluster Computing presents many new challenges to the field of computer administration. Whilst traditionally, computers on a network could be serviced manually by technicians, this approach becomes infeasible when your cluster encompasses hundreds of nodes. There is call, therefore, for new administrative tools to help us manipulate entire clusters through a single

interaction.
One approach to solving this problem is to regularly flash the same operating environment upon each node in a cluster, thus removing the need to continually ensure all nodes are pulled in to the same state [8]. Whilst this solution is elegant for homogenous nodes, it is not applicable for situations where different nodes must be setup with different environments.
An objective of OmniPanel, therefore, is to allow interaction with many instances of the same application running within different environments, via a single Control Panel widget.

4.2 Existing systems

4.2.1 Puppet

Puppet [5] is arguably the most well known IT automation package available on the market today. It is used by many organisations to simplify the administration of their servers, be there tens, hundreds or thousands of different machines located on-site or within the cloud. Primarily, Puppet is used as a tool for updating the configuration of a remote machine, for example, updating a particular library across all your hosts or amending the location to which a log file is written. Puppet allows you to define this new setup using their new declarative configuration language which can then be used as a rule-base to ensure that all remote machines comply.

4.2.2 Oracle SALT

SALT [7] (Service Architecture Leveraging Tuxedo) is an addon for the popular transaction-oriented middleware, Tuxedo. Tuxedo was originally designed to facilitate transaction processing for an application that is distributed over multiple machines and needs to support thousands of interactions each second. SALT develops Tuxedo further so that it can be used to interconnect many different systems together, first by exposing them as a web-service and then using Tuxedo to manage their communication. A web service essentially takes all the functionality provided by a back-end and represents it as a single entity on a network with which other entities can interact.

4.2.3 Domain Technologie Control

Domain Technologie Control [4] (DTC) provides a web-control panel that allows for the quick and easy provision of a server instance, be that a new SQL server, an apache host or a VPS (Virtualised Private Server). Essentially, it is a tool that allows a system administrator to lease out hosting to paying clients; DTC is installed on a high specification server machine and can be used to partition system resources in to multiple, logically seperated services owned and used by different customers.

4.2.4 cPanel

cPanel [3] is the de facto standard when it comes to web control panels. It is offered by almost all web-host providers nowadays allowing customers to tailor their web-hosting to match their individual needs. For example, cPanel allows a customer to easily create a new email account for their domain, install many common web applications such as the Wordpress blogging system and for more advanced users, even schedule automated jobs and launch ruby-on-rails applications.

Chapter 5

Ethics

5.1 Data Security

Due to the nature of this project, it is unavoidable that users will have to divulge sensitive information such as the IP addresses of their remote servers and more importantly, the credentials needed to open a secure channel with these machines. Without asking the user to supply this information, the only way OmniPanel could monitor or manipulate the state of a server would be via the use of an agent installed upon the remote host itself. This violates a functional requirement of the OmniPanel system stating that a remote host should not require modification for OmniPanel interaction.

It is essential that all data collected from users is stored in a secure manner. For each deployment, the associated host and user data is written out to a file and encrypted using DES3, the latest variant of the Data Encryption Standard. Furthermore, data is only retained for the period in which it is needed by the system - when a user deletes a deployment, the associated data file is deleted also.

5.2 Privacy

Since my system is effectively a web front-end to a number of remote services, it would be possible to monitor and log interactions to watch how clients use their applications; the commands they issue and at what times of day, for example. Storage of this information contributes nothing to the functionality of the system and would no doubt deter many clients from using the system at all. For this reason, user interactions are not logged by any components in the system.

Chapter 6

System Architecture

6.1 Database Models

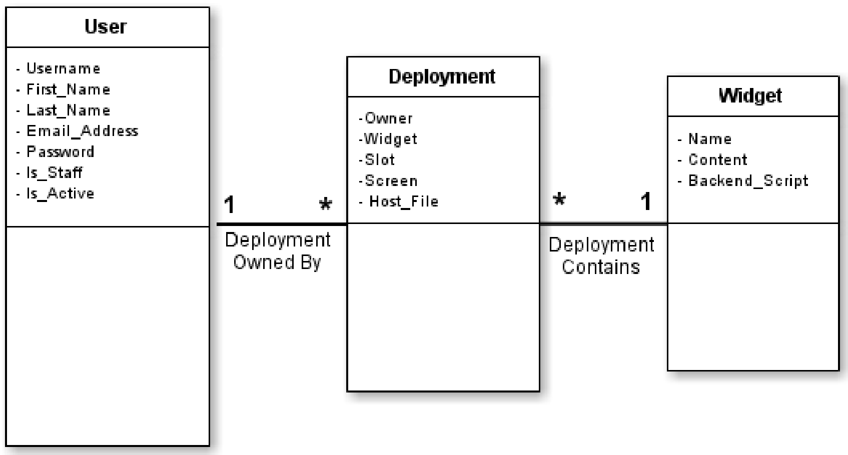


Figure 6.1: The three database models of OmniPanel

The diagram above shows the three main database models that underpin the OmniPanel system.
Any users registered with the system has their own 'User' object containing their first and last name, email address, password and access credentials.
A user owns one or more deployment; an instance of a widget that has been configured against remote host(s) and added to a slot upon the user's control page.
Finally, a widget contains a name (which is displayed in the widget library), content/template (in the form of HTML) and an associated back-end script (used to parse the server's state and process user interactions).

The Deployment Host File
OmniPanel imposes no limit on the number of hosts that can be configured against a single deployment. For this reason, it is impractical to store host information within the OmniPanel database as a single table field may not be large enough and adding a new table row for each host seems wasteful.
Instead, host information is written to an encrypted file and associated directly with a deployment instance, depicted in the diagram above as the *Host_File* attribute.

6.2 Monitoring A Host

6.2.1 Widget Placeholders

OmniPanel presents current server state to a user through the run-time injection of values in to a widget template. When defined, widgets are HTML documents that contain special place-holder strings (denoted using '!!' characters). These placeholders can then be replaced by the OmniPanel system at run-time to populate the widget template with meaningful data:

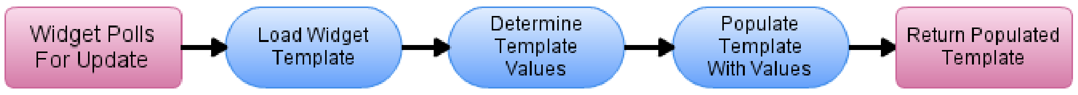

Widget Polls For Update
Load Widget Template
Determine Template Values
Populate Template With Values
Return Populated Template

Figure 6.2: Representing server state to the user

```
<b>Current Colour: </b> !!CURRENT_COLOUR!! <br />
```

Figure 6.3: Example usage of widget placeholders (frontends/ColourWidget.html)

6.2.2 Getting Template Values

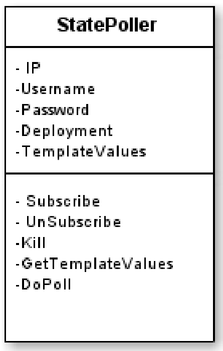


Figure 6.4: UML diagram of StatePoller object

The component responsible for provision of template values used to overwrite place-holders at run-time, is the **StatePoller** object.
A StatePoller object encapsulates an SSH connection to a remote server. It contains, therefore, an IP address, an SSH username and an SSH password.
It also stores *TemplateValues*, the latest set of template values it has generated, ready for injection in to widget content. The *DoPoll* function runs every x seconds, issueing commands to the server and processing its response to get new template values.
The *GetTemplateValues* function is used by OmniPanel to retrieve the latest set of template values.
The *Subscribe* and *Unsubscribe* functions are used to keep track of how many deployments are making use of a StatePoller object, ensuring it is not torn-down whilst it is still needed (this is explained more on the next page).

6.2.3 Multiple StatePoller Objects

Since a StatePoller object is associated with only remote host, we find that many StatePoller objects must exist simultaneously within the middleware. In order to conserve resources, StatePoller objects are spawned and torn-down as required. For instance, a deployment that has been configured against 10 remote servers will not have 10 StatePoller objects existing in parallel, instead, only one StatePoller object will be instantiated, to facilitate monitoring of the single host the user has selected in their widget at that time. If the user changes the host they wish to monitor, the old poller will be torn-down and a new one instantiated.
To further support scalability, StatePoller objects can be shared between multiple deployments when these deployments are configured against the same remote hosts. This does, however, mean we have to take extra care when tearing down resources; even though the original parent deployment may no longer be online, other deployments may still be using the resource:

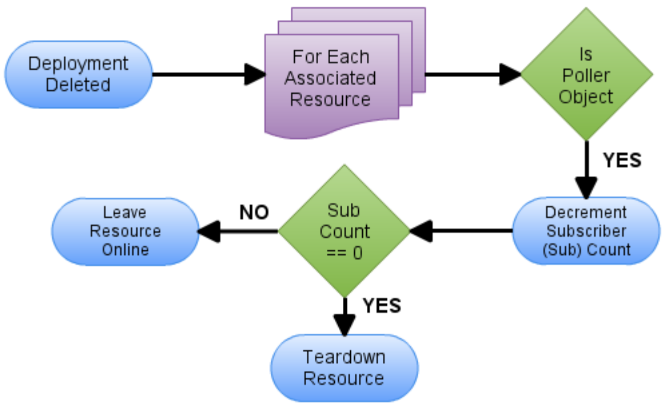


Figure 6.5: Process diagram for poller sharing. Preserve or Teardown?

6.2.4 Back-End Involvement In State Retrieval

Ascertaining the values that are to be injected in to a template is a two stage process. First, appropriate commands must be issued to a remote host and its response must be captured. Secondly, the response(s) from each command must be parsed to extract relevant information.
Since the commands that must be issued and processing that must be performed is entirely application specific, this functionality must be offloaded to an application specific back-end script.
The diagram below depicts the interaction between the OmniPanel Core, a StatePoller object and a back-end script.

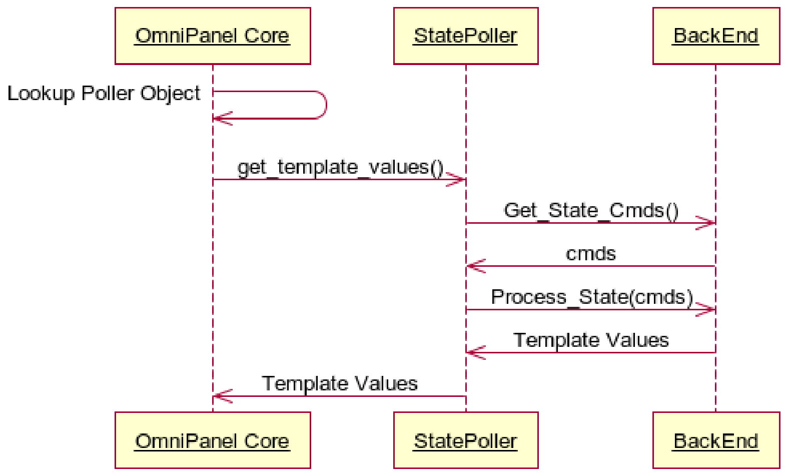


Figure 6.6: Component interactions for retrieving template values

It is important to note that for simplicity, figure 5.4 depicts synchronous interaction between all components. In reality, the StatePoller component behaves autonomously, retrieving new widget values on a regular basis. If new values were to be retrieved only on demand, the client would have to wait for a server's response to be retrieved and processed, introducing unnecessary delay.

Compulsory Back-end Functions

get_state_cmds
A compulsory function that returns a list of commands that must be executed on remote host to retrieve application state.
Input: None
Output: cmds - List of commands that must be executed on remote host.

process_state
A compulsory function that parses a server's response to status commands and returns values to be rendered in widget HTML.
Input: responses - List of server responses ordered chronologically, one list entry per instruction issued.
Output: template_values - Meaningful values extracted from server response(s), used to overwrite place-holders in front-end widget.

6.3 Manipulating A Host

6.3.1 Exposing Functionality Within A Widget

In order for a user to manipulate the state of a remote host, a deployed widget should clearly present all the functionality it provides.
For example, the widget developed to manipulate the colour of a remote computer's monitor presents the three colour options within a drop-down menu.
There are no restrictions on the HTML input elements that can be used within a widget. All that matters is that related input elements are defined within the same HTML form. When this form is submitted by the user, all of the related input data will be passed to a single function for processing.

```
<html>
  <h2>Colour Management</h2>
  <form method="post" action="interact/!!slot_id!!/">
    <input type="hidden" name="function" value="manage_colour" />
    <b>Current Colour: </b> !!CURRENT_COLOUR!! <br />
    <b>Set Colour: </b>
    <select name="NewColour">
      <option>Red</option>
      <option>Green</option>
      <option>Blue</option>
    </select> <br /><br />
    <input type="submit" value="Update" />
  </form>
</html>
```

Figure 6.7: Widgets are essentially HTML forms with a few compulsory fields

6.3.2 Routing An Interaction

Users are able to manipulate the state of a remote host by representing their desired state within a HTML form (part of the widget). When the form is submitted to the OmniPanel system, input data is routed to a back-end script for processing.

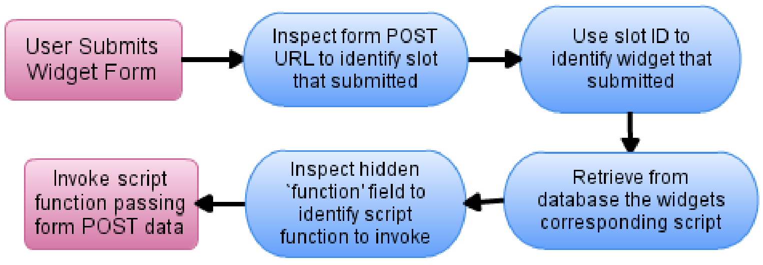

User Submits Widget Form
Inspect form POST URL to identify slot that submitted
Use slot ID to identify widget that submitted
Retrieve from database the widgets corresponding script
Inspect hidden 'function' field to identify script function to invoke
Invoke script function passing form POST data

Figure 6.8: Routing A Widget Interaction To A Back-end Function

OmniPanel determines which widget has submitted data based upon the slot in which the widget is deployed. This value is automatically injected by OmniPanel at run-time when a new deployment is made.

```
<form method="post" action="interact/!!slot_id!!/">
```

Figure 6.9: Widget definitions have a compulsory place-holder for routing

Knowing which widget submitted data allows us to identify the back-end script that must handle the interaction. Since multiple forms can exist within a single widget, we must also know which script function processes this specific form. This is defined by the original widget author within a hidden form input field.

```
<input type="hidden" name="function" value="manage_colour" />
```

Figure 6.10: A hidden form field identifies the function to process interactions

6.3.3 Generating Commands

Once a user interaction has been routed to the correct back-end function, the data they entered must be processed to generate the command(s) needed to fulfill their request.

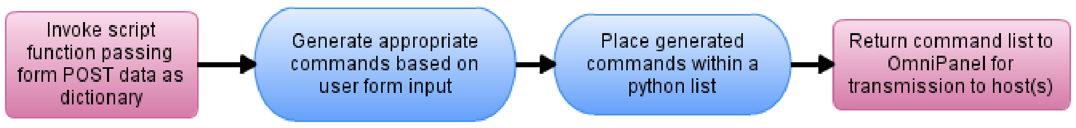

Invoke script function passing form POST data as dictionary
Generate appropriate commands based on user form input
Place generated commands within a python list
Return command list to OmniPanel for transmission to host(s)

Figure 6.11: Script functions process user input to generate application commands

The POST data uploaded from a widget HTML form is passed in to the appropriate back-end function within a python dictionary. The motivation for this is that the dictionary can easily be keyed to match the name of the input element from which the data has been extracted. For example, to access the colour that the user selected from the colour drop-down menu (see widget definition above), we simply use the following code fragment:

```
formDict['NewColour'][0]
```

Figure 6.12: Accessing form values within the back-end script

'NewColour' was the name of the HTML 'select' element defined with the widget and since this input element returns only one value, the data is available in index zero. The commands to be issued to a server are passed from the script to the OmniPanel system within a Python list object.

6.4 Issueing Commands To The Remote Host(s)

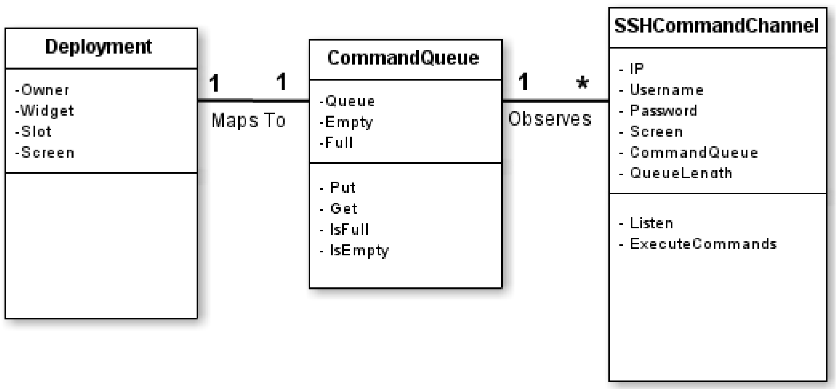


Figure 6.13: Key Components In The Transmission Of Commands

The diagram above depicts how a deployment relates to a **CommandQueue** and **SSHCommandChannel**, two components that are essential in the transmission of commands from OmniPanel to one or more remote host.
Once commands have been generated by a back-end script, OmniPanel places them within a queue of commands that are awaiting transmission.
A **CommandQueue** object is nothing more than a standard python Queue. A command queue is observed by one or more **SSHCommandChannel** object.
A CommandChannel object encapsulates a single SSH connection to a remote host. For a multi-host deployment, therefore, we will spawn multiple channel objects.
All channel objects belonging to a single deployment monitor the same command queue. This reduce computational complexity and conserves system resources as having seperate queues would be needlessly expensive. One complication of having multiple objects operating over the same queue is that a channel cannot remove an item from the queue (in case it hasn't yet been seen by another channel).
Each channel object therefore holds the size of the queue the last time it was inspected within an attribute called *QueueLength*. If a queue has grown in length since the last inspection, new commands must be issued to the remote host.

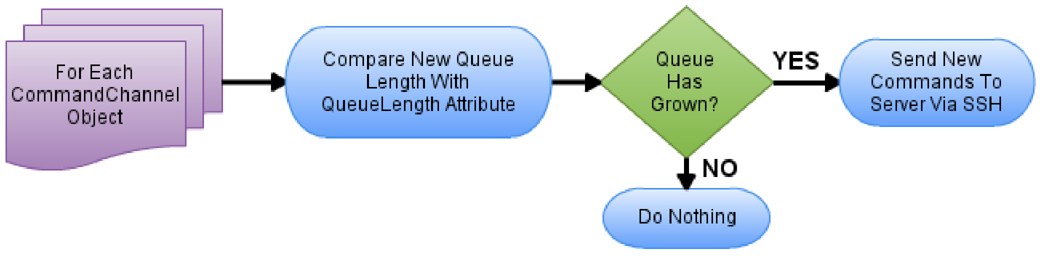

For Each CommandChannel Object
Compare New Queue Length With QueueLength Attribute
Queue Has Grown?
YES
Send New Commands To Server Via SSH
NO
Do Nothing

Figure 6.14: Key Components In The Transmission Of Commands

6.5 Host Discovery

6.5.0.1 Sending A Beacon

The autodiscovery feature of Omnipanel allows for hosts on a local network to be automatically discovered and configured against a deployment, without interaction from the user.
The system is underpinned by local hosts regularly transmitting a specialised message via UDP multicast, this special message is referred to as a beacon.
In order for a host to transmit a beacon message, a small application must be installed on the host itself. This application is referred to as a Beacon Agent (See Appendix D).

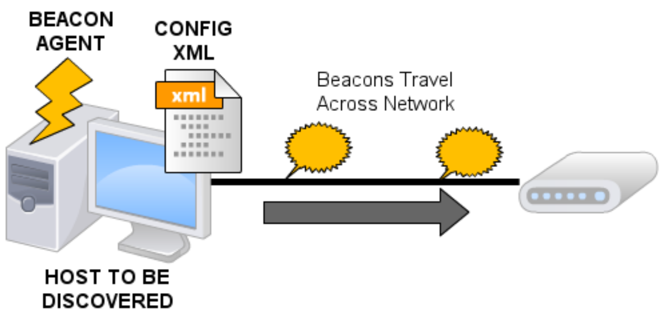


Figure 6.15: A Beacon-Agent sends beacons across the network

Each beacon message transmitted from a client contains an identifier string, the value of which can be altered within the application configuration file.
The identifier payload of a beacon message makes it distinguishable from other beacon traffic, allowing different subsets of hosts to beacon simultaneously and yet be discovered as separate groups.

6.5.1 Receiving A Beacon

Omnipanel contains two components that are integral to the host discovery mechanism:
A **BeaconListener** object binds to a specific multicast address and port, specified in the Omnipanel settings file, and listens for beacon messages. Any Beacon messages that are received are reported to a **BeaconManager** object.

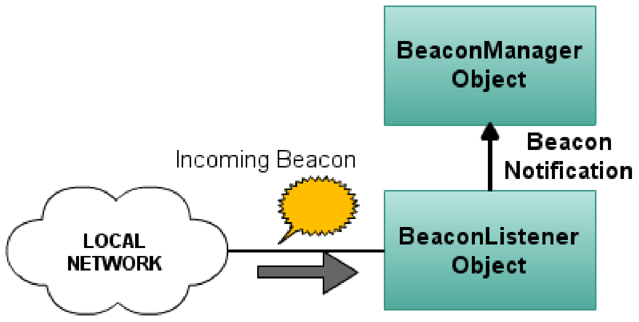


Figure 6.16: Incoming beacons are reported to a BeaconManager

The **BeaconManager** object keeps a list of all the IP addresses that beacon with a specific identifier.
It is important that an IP address is not appended to a list more than once else we run the risk of configuring a single deployment against the same host multiple times:

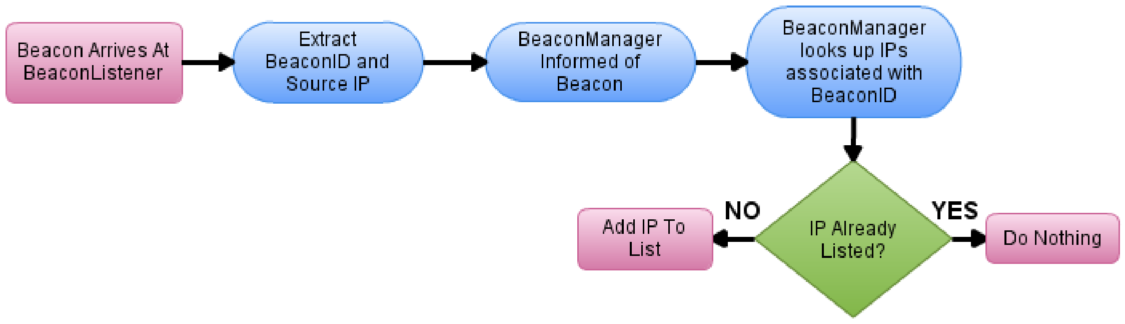

Beacon Arrives At BeaconListener
Extract BeaconID and Source IP
BeaconManager Informed of Beacon
BeaconManager looks up IPs associated with BeaconID
IP Already Listed?
NO
Add IP To List
YES
Do Nothing

Figure 6.17: Process diagram avoiding duplicate IP entries.

6.5.2 Reporting Discovered Hosts

The final step in the autodiscovery process is for a list of discovered IP addresses to be returned to the user.
This occurs when a user makes use of the auto-discovery tool within their HTML front-end. Rather than manually entering the IP addresses of all hosts to be controlled, the user can specify a beacon identifier to search for. This identifier value is transmitted via AJAX to the OmniPanel core where is then used as a lookup key on the **BeaconManager** object. The manager will return a (potentially empty) list of IPs that have been known to beacon with that identifier. This list is then forwarded to the client for display in their table of configured hosts.

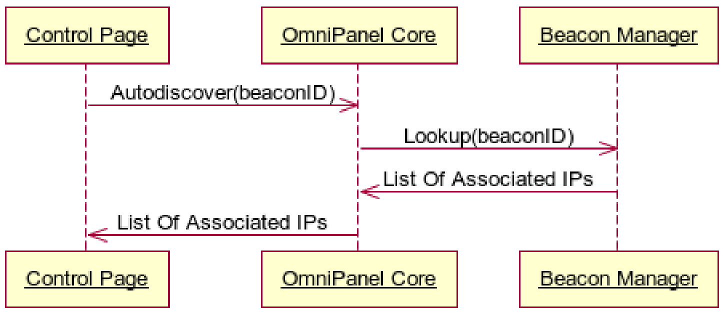


Figure 6.18: Component interactions for automated host discovery.

6.6 Client Heartbeats

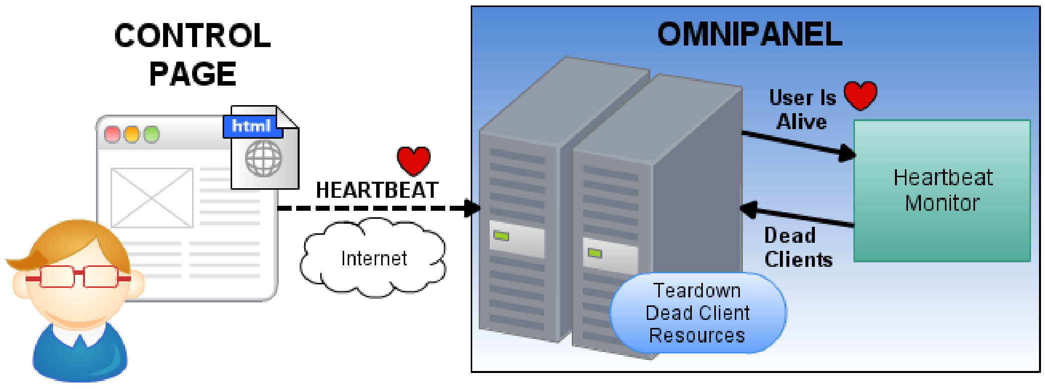


Figure 6.19: Overview of HeartBeat Mechanism

Omnipanel makes use of a client heartbeat mechanism to determine whether a user is still using the system or whether they have navigated away from their control panel. This information is useful as we want to tear-down any Omnipanel resources that have been provisioned for that user whenever they are no longer needed.
Whilst the Omnipanel Control Page is open within a user's browser, it will transmit a heartbeat message every 30 seconds. Upon receipt at the Omnipanel web-server, the heartbeat is forwarded to the HeartBeat Manager component where we record the timestamp of that heartbeat.
The HeartBeat Manager runs a scan every 40 seconds using timestamps to detect when a user has missed a heartbeat. It allows a period of grace whereby a single heartbeat can be skipped without resource tear-down - this allows for possible network issues where the heartbeat message may simply have been lost enroute. If two heartbeats are missed, however, the HeartBeat Manager informs the OmniPanel core that the user has closed their session and that resource tear-down should begin.

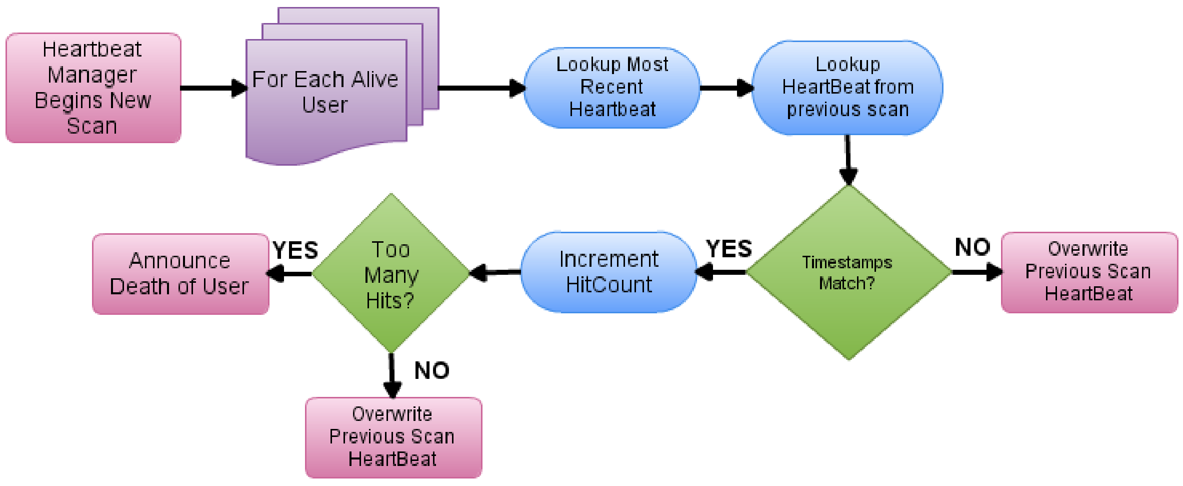

Heartbeat Manager Begins New Scan
For Each Alive User
Lookup Most Recent Heartbeat
Lookup HeartBeat from previous scan
Timestamps Match?
NO
Overwrite Previous Scan HeartBeat
YES
Increment HitCount
Too Many Hits?
YES
Announce Death of User
NO
Overwrite Previous Scan HeartBeat

Figure 6.20: Determing Whether A Client Has Died

6.7 Designing For Scalability

OmniPanel has been written with scaleability in mind throughout. With the potential for OmniPanel to be deployed on a central web server and made available to the general public as a whole, it is essential that it can scale to meet demand.
Allow us to discuss some of the deliberate design choices taken to promote scaleability within Omnipanel.

6.7.0.1 On-Demand Resource Spawning

OmniPanel adopts an on-demand approach to resource spawning, only provisioning resources to a user when they are definetely required. For example, when a user creates a multi-host deployment, OmniPanel does not open SSH connections to all associated hosts. Instead, a *StatePoller* component is spawned to communicate with the single host that the user has chosen to monitor via their widget at that point in time. Furthermore, if the user chooses to change the host they wish to monitor, the existing *StatePoller* component is torn-down prior to spawning of a new one. With the current limitation on the number of available deployment slots, it is therefore only possible for each user of the system to have a maximum of four *StatePoller* components running within the system.

6.7.0.2 On-Demand Resource Teardown

Any components that are spawned within OmniPanel are directly associated with the user to whom they belong. If that user logs out from the system or navigates away from the ControlPage for longer than 60 seconds, any associated components are automatically torn-down allowing OmniPanel to reclaim the resources they consume. Furthermore, a subset of the users components may be torn-down whenever a user deletes a deployment, allowing for reclaim of any *StatePoller* and *CommandChannel* objects, for example.

6.7.0.3 Resource Sharing

Whenever possible, OmniPanel tries to prevent the spawning of a duplicate components by identifying when a single component can be used to service multiple requests. For example, if a user has two deployments that both interact with the same remote application on the same remote host, it is possible for each of those deployments to use the same *StatePoller* object. Now, contacting the remote host and parsing of it's response need only be performed once, with new widget content being pushed to both deployments. Sharing of resources not only conserves system resources, it also reduces computational overhead with response processing being performed once rather than twice.

6.7.0.4 Synchronisation Of Polling Processes

As has already been discussed in the design section, the monitoring of a remote applications state is underpinned by two distinct polling processes; one between the user and OmniPanel and a second between OmniPanel and the remote server. OmniPanel uses timestamps to measure how often the front-end polls for an update and uses this value to adjust its own internal polling frequency. Use of this mechanism ensures that OmniPanel is not wasting resources polling for remote state when this state is never actually transmitted to the user.

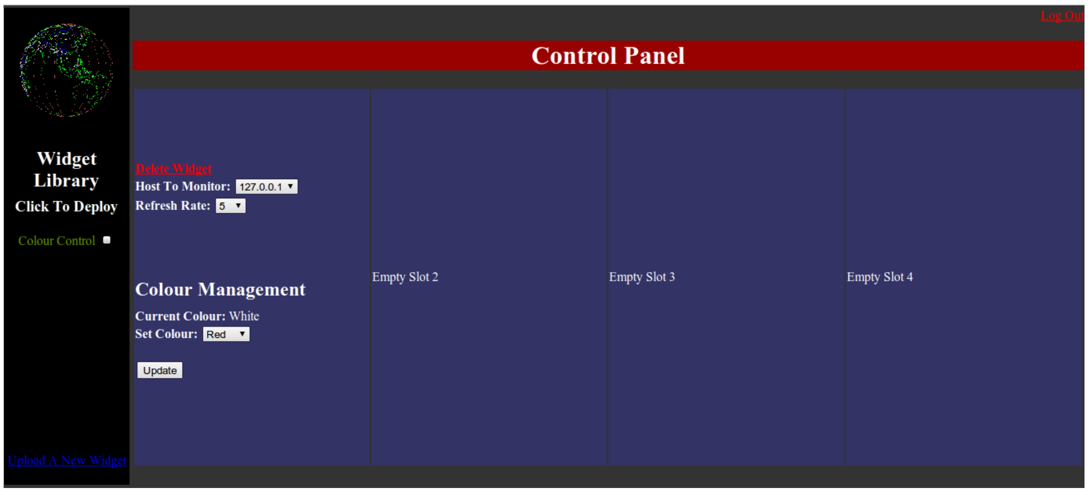
6.7.0.5 A Global Widget Library

Whenever a new front-end back-end pairing is uploaded to OmniPanel, the front-end HTML is written in to the Django database and the back-end script is output to disk. For this reason, the library of available widgets is global across the OmniPanel system meaning that when new functionality is uploaded by a single user, it can be seen and deployed by all system users. If widget libraries were local to a single user only, we would no doubt have significant data duplication on disk as many users upload and use the exact same functionality.

Chapter 7

User Manual

The entire user interface is exposed within a HTML web browser.



Host To Monitor: 127.0.0.1

Refresh Rate: 5

Colour Management

Current Colour: White

Set Colour: Red

Figure 7.1: The User Control Panel

The widget deployment process by the user selecting the widget they wish to deploy from a library of available widget displayed on the left of their Control Panel.

7.0.1 Deploying A Widget

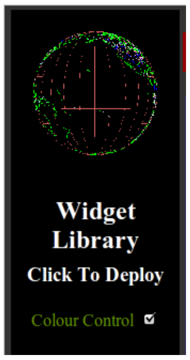


Figure 7.2: Select A Widget From The Library

Next, a configuration menu is displayed to the user allowing them to input the information needed for a successful deployment. Specifically, the user must provide the IP addresses and credentials needed to access remote machines. This information can be input by the user via a variety of mechanisms:

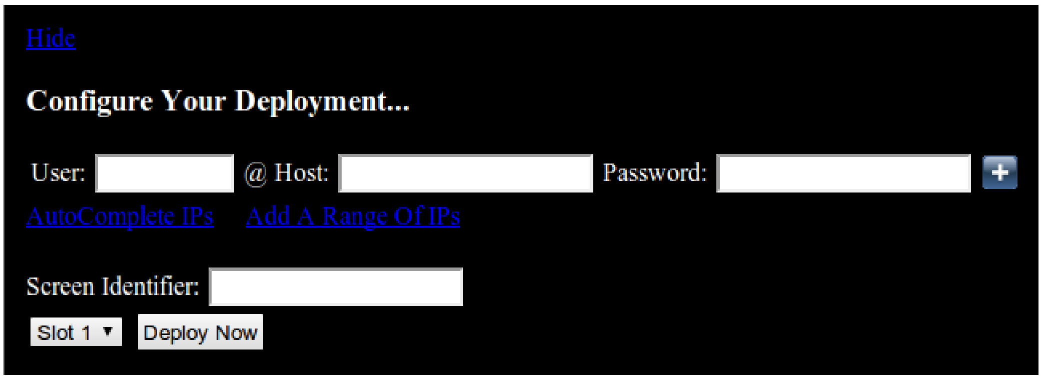


Figure 7.3: The Deployment Configuration Window

7.0.1.1 Manual IP Input

The user is able to manually input IP and SSH information within a HTML table in the configuration menu. Initially, fields are only visible for configuration of a single host as only one host is required for use of the system. This also helps to keep the configuration menu as de-cluttered and simple as possible. Additional hosts can easily be configured by simply clicking the '+' icon which will reveal new input fields for the user to complete.

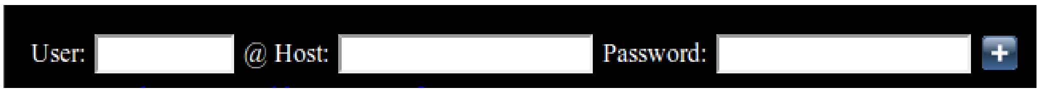
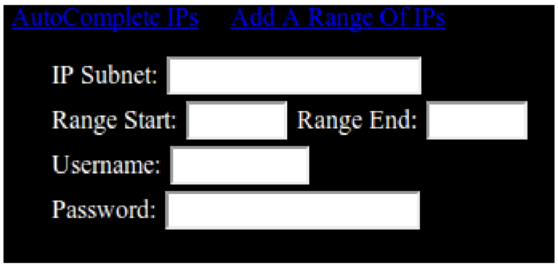


Figure 7.4: Manual IP Input Table

7.0.1.2 Inputting A Contiguous Range of IPs

A common use-case is for the user to want to control multiple hosts within a range of IP addresses all on the same subnet. Imagine, for example, a scenario where a lecturer wants to trigger the playback of a video on each student computer within a lab. This use-case is covered by the option to 'Add A Range Of IPs', a link on the page that when selected, reveals an additional deployment dialogue to the user (figure 6.5 over page):
The dialogue prompts the user to enter the common portion of the IP address range (The Subnet) along with the lowest and highest number hosts within that range. Finally, the user must provide the username and password needed to access each machine in the range. It is assumed that all machines share the same login credentials as they are all in the same physical location, under the same administration and probably share the same authentication mechanism.

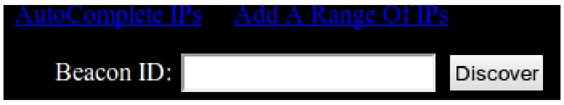


IP Subnet:			
Range Start:		Range End:	
Username:			
Password:			

Figure 7.5: Inputting A Range Of IPs

7.0.1.3 Automatic IP Discovery

Finally, rather than manually entering host IP addresses or specifying a contiguous IP address range, the user can use an auto-discovery mechanism to find IP addresses of hosts advertising themselves on the local network. This allows the user to quickly configure a multiple host deployment without the need for manual IP input even when the hosts don't occupy a contiguous range of addresses.

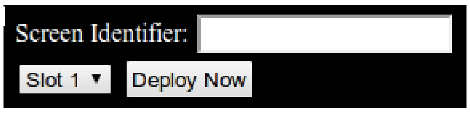


Beacon ID:

Figure 7.6: Host AutoDiscovery Tool

Use of the auto-discovery tool simply involves the user specifying a beacon identifier - a word or string that the system should look for on the network. Any hosts that are found to be broadcasting this identifier are detected by the middleware and their IP address is automatically appended to the user interface. Forwarding to the user allows us to verify that the address is correct and also allows them to the enter the username and password needed to open connection with each host.

7.0.1.4 Common Options



Screen Identifier:

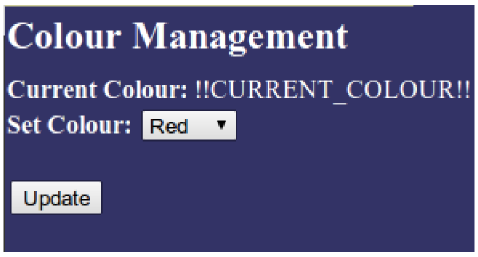
Figure 7.7: Screen identification and Slot selection

Regardless of how the user decides to input host address and access credentials, there are two additional fields that must be completed prior to deployment. Firstly, the user must select from a list, the **slot** in to which the widget is to be deployed. At present, the user control page allows a user to have only four simultaneous deployments though this could easily be expanded at a later date. The user must decide which slot to deploy to, bearing in mind that deploying in to an occupied slot will delete the previous deployment.
Secondly, the user must provide the identifier of the **screen** session that their target service or application runs within. As has already been described in the project

introduction, it is assumed that all applications run within screen sessions due to their multiplexed and persistent nature.

7.0.2 Monitoring A Host

Once a widget has been deployed, it can be used to monitor the state of one or more remote hosts.
The state of a server is presented to the user by over-writing placeholders within the template. The diagram below shows a template before placeholders have been overwritten:

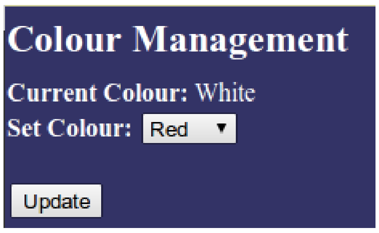


Colour Management

Current Colour: !!CURRENT_COLOUR!!
Set Colour: Red

Figure 7.8: Unpopulated Widget Template

Once the middleware polls the remote server and it's response is processed, replacement template values are used to populate the template appropriately:



Colour Management

Current Colour: White
Set Colour: Red

Figure 7.9: Populated Widget Template

7.0.2.1 Adjusting A Monitor

Widgets will automatically update themselves with new content on a regular basis so users can view the latest state simply by watching their control page. In order to conserve resources and maintain a strong user experience, it is possible to adjust how often a widget polls for an update.
Furthermore, the user is able to select which remote host they wish to monitor (assuming a deployment has been configured against multiple IPs).
Both the refresh rate and the IP to monitor can be set by the user by manipulating these options at the top of each widget: Each widget header also gives the option to delete a deployment thus freeing up a slot on the user's control page and tearing down all associated resources in the middleware.

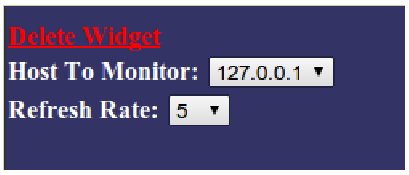

Delete Widget
Host To Monitor: 127.0.0.1
Refresh Rate: 5

Figure 7.10: Adjusting monitor refresh rate and remote host

7.0.3 Manipulating A Host

Manipulating the state of a host is a simple as representing your desired state within the HTML widget and submitting it to OmniPanel. The screenshot below shows how the proof-of-concept 'Colour' widget exposes the three possible states that can be selected by the user. The user selects the colour they wish to set on the remote host and submits it using the 'Update' button.

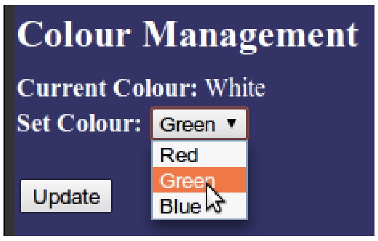

Colour Management
Current Colour: White
Set Colour: Green

Figure 7.11: Inputting Desired State For Remote Application

7.0.4 Uploading New Functionality

One of the features that makes OmniPanel so unique is the user's ability to extend the functionality of the system via the upload of new widgets and back-end scripts. The upload dialogue is accessed via the 'Upload A New Widget' link at the bottom of the widget library.

Figure 7.12: Display The Upload Dialogue

Clicking the link reveals the following upload dialogue:

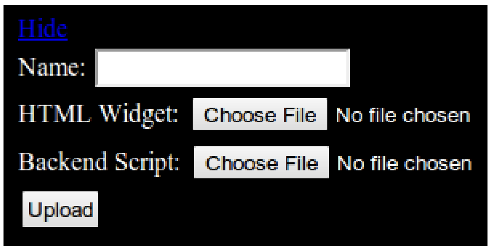

Hide
Name:
HTML Widget: Choose File No file chosen
Backend Script: Choose File No file chosen

Figure 7.13: Uploading New Functionality

The 'Name' field is used to identify the newly uploaded widget within the library.

The two file upload fields allow the user to upload a HTML file containing the front-end widget and a python script containing the back-end functionality.

Chapter 8

Software engineering techniques and processes

8.1 Agile Programming

In the early stages of this project, a basic system architecture was drafted. The architecture defined the different components of the system, the functionality they would expose and the different ways they would interact. Very quickly, it became apparent that this system was simply too large and too complex to design entirely upfront. Instead, it was decided that breaking down the project in to a series of *milestones* would be a better design approach.
Typically, each milestones marked the realisation of a particular feature within the system such as the ability to deploy a widget within a slot or the ability to issue a command to a remote server. This process of breaking down your system in to set of features and implementing them one at a time is known as Feature Driven Development. Feature Driven Development is just one of the many methods that are considered Agile. Agile means that no plans are set in stone before development begins, strongly contradicting the traditional water-fall model where you always fully plan and design prior to implementation. The benefit of Agile development is that it helps you to realise shortcomings in your design at an earlier stage, leaving you more time to reconsider and redevelop your codebase. It also helps you to discover the limitations of libraries and technologies you have chosen to use - on several occasions, the OmniPanel codebase was refactored not due to a poor design choices but simply due to limitations imposed by third-party tools.

8.2 Version Control

Before work began on writing the OmniPanel codebase, a new version control repository was setup for the OmniPanel project. The use of version control is essential for any large-scale project as it protects you not only against a loss of code should your local copy be lost or corrupted, but also helps you to rollback any detrimental code changes such as the accidential introduction of a bug in your system. The SubVersion version control system was chosen due to familiarity from previous projects and the fact that it integrates so easily with the Ecliple IDE. SubVersion has, in the past, caused problems when two different people try to commit changes on the same file but this problem should never arise when a project has only one code contributor. The functionality of SubVersion with augmented with the Trac tool [1], an open source project that allows you to explore your project repositories via a web front-end. Trac allows you to see a Timeline of your code commitments, colouring lines in the code that have been modified since the last commitment as well as hosting a project wiki, bug tracker and RoadMap of past and future *milestones*. Whilst many of these features are aimed at projects that have many contributors, trac proved invaluable helping us to visualise progress through the project.

8.3 Coding Standards

One of the biggest problems faced when developing in an agile manner is a constantly evolving code-base. The way in which a functionality is exposed or even the component responsible for exposing that functionality can change due to amended design choices or limitations imposed by system libraries. To cope with this, one must employ a very disciplined approach to coding. If the way in which a function is exposed changes, adding or removing parameters to a function call, for example, it is wise to consider all the places where that function can be called from and update that code immediately. Enforcing this behaviour not only helps to prevent run-time exceptions where nonsense arguments are passed, but also helps developers to get a bigger picture of how their system now functions. It is often the case that studying when and where functions are called, leads to the development of an optimal system architecture where functionality is placed near to the components that need it most (thus reducing complexity and often improving performance).

There are several additional techniques that are fundamental to producing a high standard of code. Firstly, one should always be able to see what a section of code is doing without having to compile and run the source. This can usually be achieved through the use of meaningful variable names and the occasional comment to explain a particularly convoluted conditional statement. Perhaps even more important is the correct use of indentation and formatting so you can see which code runs as a result of a condition or which functions are called repeatedly within a loop. Our decision to use Python as a core programming language forced us write well formatted code as python parses the source based on indentation, rather than on the use of brackets like in languages such as Java.

Finally one must take care to remove redundant code from your code-base. Adapting to a constantly evolving design is unfortunately not as simple as just updating function calls, developers must also consider whether any of the surrounding codebase is now surplus to requirements. Failure to do so not only wastes resources (you don't want redundant code executing on a low specification, embedded device) but also serves to confuse any engineers in the future who are trying to understand your source.

[1]http://trac.edgewall.org/

8.4 Development Logs

Multiple logs were kept throughout the development process to document design decisions and any problems that were encountered along the way.
The motivation for these logs was two-fold; Firstly, with so much development needing to be done and so many components to keep track off, it was unlikely that the justification for all design decisions could be recalled at the end of the project. Since this information is of value within this report, logs were kept so that key design decisions could be explained clearly.
Secondly, in the early stages of the OmniPanel project, a lot of time was lost resolving coding problems that later turned out to be very trivial mistakes. Even more frustratingly, a few of the mistakes such as an incorrectly formatted import statement were made more than once costing just as much time on the second occasion as they did the first. For this reason, a log entry was created for mistakes that were deemed easily repeatable in the hope of finding a quick solution later in the development process.

8.5 Testing

Due to the highly modular design of my system, it was essential that all components be tested on a regular basis. Failure to test the behaviour of a single component may well compromise the entire system and worse still, makes it very hard to ascertain exactly what went wrong once all the components have been integrated together. Due to the complex interactions between components and the fact that many functions pass back python objects rather than literal values, OmniPanel did not lend itself to comprehensive unit testing. Instead, great care was taken to consider dangerous edge-cases when writing the code, ensuring that no unsafe assumptions were being made. In any scenario where something could potentially go wrong, failure to retrieve a deployment object from the database, for example, code is encapsulated within python 'try' and 'except' statements. This allows for the printing meaningful error messages to the console or to system logs before gracefully returning from a function. Alongside the inclusion of meaningful error messages, a context-driven approach to testing was adopted throughout. This meant that once enough components had been developed and integrated together, the system was tested through use of its functionality (monitoring and controlling remote machines). The Senior Honours lab was the perfect testing environment due to the high number of machines free to use at any time.
Finally, one of the most important factors influencing the uptake of the OmniPanel system is the question of scalability. At the end of Chapter 6 we describe exactly how the system was designed and implemented with scalability in mind. Unfortunately, reassuring oneself that true scalability has actually been achieved is no easy task. The obvious solution is to stress-test the system, monitoring and manipulating many different hosts in parallel. Unfortunately, even with all the lab machines at our disposal, there are insufficient resources to really push the system to its limits. Instead, all we could do is re-assure ourselves that the measures taken to facilitate scalability were working as expected - this includes the use of console output or system logging to show that middleware resources were being freed up once they were no longer needed by a deployment. Furthermore, system logs indicate when the system has

located a resource that can be shared between different deployments, for example, when two deployments are setup to monitor the same remote host, we only need to poll for state once and this can be used to update both front-end widgets.

8.6 Documentation

To encourage future development of the OmniPanel core, the OmniPanel codebase has been documented using the industry standard python ‘docstring’ methodology. A docstring is simply a string enclosed within triple speech marks that is appended under each class and function declaration.
Class documentation describes the role of the class within the overall system, perhaps relating it to a real-world entity.
Function documentation describes the role of each single function within a class, including any parameters it takes and any values it returns.
Files containing docstrings can be processed using the command-line ‘pydoc’ application. This tool parses source-code to display human-readable documentation directly within the terminal or to generate HTML files.
Further to the use of pydoc docstrings, individual python comments are included throughout the codebase to clarify ambiguous sections of code.

Chapter 9

Implementation

9.1 Front-end

9.1.1 HTML

The HTML front-end exposed by Omnipanel has been mostly hard-coded in HTML format. The code was written from scratch within a standard linux editor supporting HTML syntax highlighting.
The front-end exposes two interfaces to the user; the login screen and the user's personalised control page. The former interface is completely static containing two input fields, a header and an animated gif. The personalised control page, however, contains elements that must be dynamically injected in to the web-page (delivering the correct widgets to the correct user). This dynamic injection is achieved using the Django framework which I will return to when discussing implementation of OmniPanel core.

9.1.2 CSS

Cascading Style Sheets (CSS) have been used to apply styling to the aforementioned HTML pages.
Key elements of each page such as the widget library, the four widget slots and the configuration drop-down are all contained within seperate HTML DIV sections on the page. Partitioning the document in this way allows for easy application of different styling rules to different elements on the page.

9.1.3 JavaScript and JQuery

JavaScript is the industry standard programming language for implementing client-side functionality within a web-browser. Offloading computation to the client is desirable in our system as we wish to promote scaleability, minimising the middleware resources that are consumed by each client.
JQuery is merely an extension to JavaScript, a library that presents a more conise and well-supported API to the javascript engine.
JQuery is leveraged on many occasions within Omnipanel. It is used to manipulate the Document Object Model (DOM), allowing us to dynamically create and manipulate HTML elements on the page. It is also used to trigger AJAX interactions between the client and OmniPanel core.

Allow us now to discuss these use-cases in more detail.

9.1.3.1 Growing The Host Input Table

JQuery is used to dynamically add new fields to the host configuration table whenever the user clicks on the '+' icon. This allows the user to manually input the details of an unlimited number of remote hosts, all to be monitored and manipulated by a single deployment.

```
var newRow = jQuery('#IPTable tr:last').clone(true)
jQuery(newRow).hide();
jQuery(newRow).insertAfter('#IPTable tr:last');
jQuery(newRow).slideDown();
```

The code above demonstrates the use of a JQuery **Selector** to easily select the final row element of the table named 'IPTable'.
It also demonstrates how this easily be cloned, hidden, appended to the end of the table and then animated in to display for the user.

Other Uses Of JQuery:

9.1.3.2 Front-end Polling For Updates

As already discussed in the design section, retrieving state from a remote server is a complicated process. It begins, however, with a single poll from a front-end widget to OmniPanel system.
Below is a process diagram depicting the role of JQuery in the polling process:

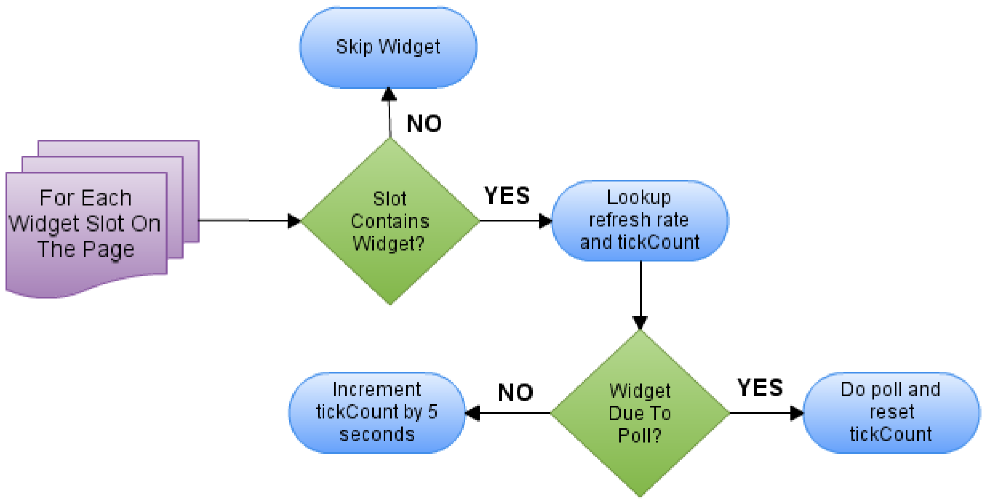

Skip Widget
NO
For Each Widget Slot On The Page
Slot Contains Widget?
YES
Lookup refresh rate and tickCount
Increment tickCount by 5 seconds
NO
Widget Due To Poll?
YES
Do poll and reset tickCount

Figure 9.1: JQuery decides whether a widget should poll for update

9.1.3.3 Front-end Heartbeat

JQuery is used at the front-end to repeatedly send a 'heartbeat' message to OmniPanel every 30 seconds. Since this instruction will only continue to execute whilst the end-user has the control page loaded in their browser, the heartbeat message signals to OmniPanel that the system is still in use. When heartbeat messages are no longer being received from a client, we can assume they have navigated away from the page and that their associated system resources can be torn-down.

9.1.3.4 Autodiscovery of hosts

JQuery is used in two different ways within OmniPanels host auto-discovery mechanism. Firstly, it is used to retrieve the beacon identifier entered by the user and forwards this to the core. Secondly, it is used to parse the list of IP addresses returned from OmniPanel, unpacking them from the JSON [1] format and appending them to the list of configured hosts at the front-end.

9.1.3.5 Toggling the visibility of DIV elements

Omnipanel makes extensive use of a JQuery script called 'Animated Collapse' [2]. This script leverages JQuery to simplify the process of showing and hiding DIV elements on a HTML page. It is this script that dynamically toggles the visibility of the host configuration drop-down as well as sub-menus within the drop-down window.
The 'Animated Collapse' script is entirely the work of the original author(s) cited on the dynamic drive website. The work is not my own.

[1] http://www.json.org
[2] http://www.dynamicdrive.com/dynamicindex17/animatedcollapse.htm

9.1.4 DAJAX (Django AJAX)

Asynchronous Javascript And XML (AJAX) is a generic term often applied to technology that allows for asynchronous communication between web browser and server. AJAX technologies allow for the automatic updating of page content without the need to refresh the page in its entirety.
This type of behaviour is desirable within OmniPanel for the automatic update of widget content and also for the client-OmniPanel interactions that you wish to hide from the user (such as heartbeats).
Unfortunately, AJAX interactions are often very complicated to code from scratch as one must develop a server-side script, a function to marshall data in to XML for transmission and a client-side javascript function to present the data on the web-page.
DAJAX [3] (Django AJAX) is a Django (See Middleware, Django) library that simplifies the process of incorporating AJAX interactions within a Django project.
Through interactions with special client-side and server-side objects, implementing AJAX interactions becomes a trivial task.
As an example, allow us to study how DAJAX is used within the host auto-discovery tool.

9.1.4.1 Host Auto-Discovery

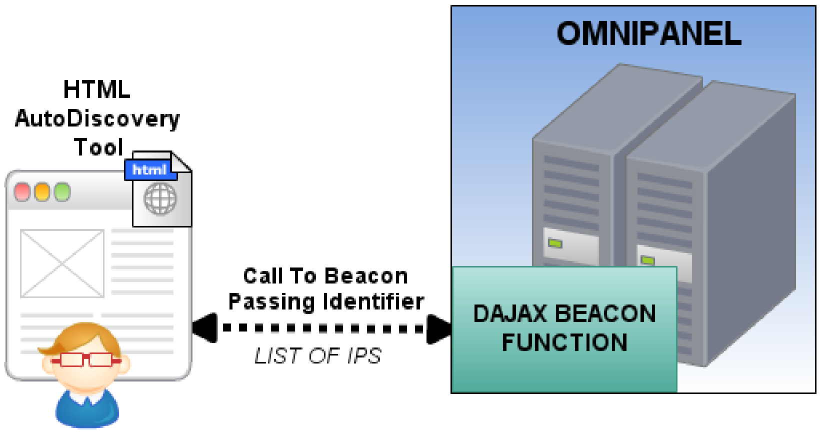


Figure 9.2: TESTING

HTML Integration
The first step in the discovery process is the transmission of the user-supplied Beacon Identifier to the OmniPanel system. This is achieved through a call to the DAJAX function 'beacon', passing the identifier as an argument.

```
Dajaxice.apps.ControlPage.beacon(Dajax.process,{'beaconID':'hello'})
```

The Dajaxice element with which we interact is a special JavaScript object, created and embedded in the HTML at run-time by the DAJAX library. The object exposes a function called 'beacon' because we have declared the function beacon within a special Django *ajax.py* file (see next page). Any arguments to be passed to the function are presented within the JSON [4] notation.

[3]http://www.dajaxproject.com/
[4]http://www.json.org

The DAJAX Function
Below is the implementation of the 'beacon' function written with the *ajax.py* file located in the *apps/ControlPage* directory

```
@dajaxice_register
def beacon(request,beaconID):
    dajax = Dajax()
    payload = getIPs('beaconID')
    dajax.add_data(payload,'unpackerFunction')
    return dajax.json()\begin{figure}[h!]
\centering
\includegraphics[scale=0.6]{images/diagrams/dajax.png}
\caption{TESTING}
\end{figure}
```

The code above demonstrates how values passed from the client as JSON arrive at the function as runtime arguments.
We see use of the DAJAX 'add_data' function to specify data for transmission to the client and also to identify the client-side function (unpackerFunction) that should be used to unpack the data.
Finally, DAJAX trivialises the conversion to JSON prior to transmission across the network.

Other Uses Of DAJAX:

9.1.4.2 Client Heartbeat

The DAJAX library is used to transmit the heart-beat message from the client to the middleware, indicating that they still have the control page loaded in their browser. This message leverages AJAX so that heartbeat transmission is invisible to the user. Transmission of a synchronous request would require a reload of the entire control page for each heartbeat message that is sent.

9.1.4.3 Updating Widget Content

The DAJAX library makes it possible for widget content to be automatically updated without the need for reloading of the entire control page. Widgets on a page poll the middleware on a regular basis (triggered using JQuery) and receive a populated widget template in response. This populated template is then set as the widget content.

9.2 Middleware

9.2.1 Pycrypto

The functionality of OmniPanel relies upon the user providing sensitive information about their remote hosts including the SSH credentials needed to access them. It is essential that this information be stored in a secure manner to protect user's systems from attack should OmniPanel somehow be breached.

Pycrypto [5] is a python library that simplifies the process of data encryption. It provides implementations of the popular encryption standards such as AES, Blowfish, RC5 and DES.
Below is a fragment of code in which the pycrypto library is used to encrypt the contents of a deployment host file, prior to writing to disk.

```
    plaintext = get_sensitive_data()
    des3 = DES3.new(settings.DES3_KEY,DES3.MODE_ECB)
    ciph = des3.encrypt(plaintext)
    hostFile.write(ciph)
    hostFile.close()
```

Pycrypto provides a DES3 object, capable of applying the third variant of the Data Encryption Standard (DES) algorithm on sensitive information. The DES3 object is instantiated with a secret key used for encryption/decryption and also a mode of operation.
Encryption is performed by the DES3 'encrypt' function, passing our sensitive plaintext as an argument and returning the ciphertext as an output. The encrypted ciphertext can then be written to disk.

9.2.2 PXSSH

OmniPanel uses SSH commands to issue commands to remote host(s). The commands issued may be part of the polling process used to retrieve server state or may be issued to manipulate the state of a server in response to a users widget interaction.
PXSSH [6] is a python SSH library used with OmniPanel to setup and use SSH connections.
The code below demonstrates the use of PXSSH to setup an SSH connection, issue the 'ls' command and store the servers response.

```
ssh = pxssh.pxssh(timeout=0.5)
ssh.login('127.0.0.1','user','password')
ssh.prompt()
ssh.sendline('ls')
response = ssh.before()
```

[5]http://www.pycrypto.org
[6]http://pexpect.sourceforge.net/pxssh.html

9.2.3 Django

Django [7] is a web application framework that leverages the Model View Controller paradigm to support rapid web development.
Many features of Django make it the obvious choice of framework to underpin OmniPanel:

Database Integration
In the design section we discuss three database tables, User, Widget and Deployment, that will be used to store persistant state of the OmniPanel system.
Django makes database integration a trivial task will full support to create tables, write new data in rows and to retrieve data using queries.
Creating a new database table involves the definition of a Django 'model' as shown below:

```
class Widget(models.Model):
  name = models.TextField(max_length=20)
  content = models.TextField(max_length=1000000)
  backend = models.TextField(max_length="1000",default="backends/default.py")
```

This will create a database table with 3 varchar columns of different size and default value.
Adding an entry within the newly created table is as simple as instantiating a Widget object and calling the 'save' function upon it.

```
w = Widget(name='My Widget',content='Hello World',backend=myBackEnd.py)
w.save()
```

Retrieving values from the database is equally trivial as Django can build appropriate database queries on the users behalf:

```
retrievedWidget = Widget.objects.get(name='My Widget')
```

The Templating System
A lot of OmniPanel's functionality depends upon the system's ability to dynamically inject values in to a web-page at run-time. For example, when a user loads their Control Panel, content must be injected to each of their widget slots depending on whether that slot contains a deployment.
Django provides a templating mechanism that makes it easy to inject run-time data within a HTML page. Template definitions contain standard HTML and optionally some Django specific constructs such as value place-holders, conditional statements and data iterators. Prior to transmission to the client, data can be passed to a template where it will be operated upon by these special template functions to populate the page with meaningful values. For example, the code below shows how a template iterator is used to process a list of available widgets (retrieved from the database) and display them within the widget library:

```
{% for widget in widgets %}
  {{widget.name}} <input type="checkbox" name="{{widget.id}}" /><br/>
{%endfor%}
```

[7] https://www.djangoproject.com/

Chapter 10

Testing

10.1 Testing Methodology

Testing the OmniPanel system involves the testing of both it's functionality and it's performance (since performance relates directly to scalability).
To aid in the testing process, a trivial widget and back-end script were developed to interact with a java application running upon remote host(s).
The role of the Java application (See Appendix A) was to simply present a full-screen window on display of the computer it runs upon. The colour of this window could then be manipulated via the issue of commands in to the application command-line, thus making it possible to remotely interact with the application over SSH.
The task of monitoring and manipulating screen colours was chosen as it lends itself well to the testing process. It is easy to compare the reported state of a machine against it's actual state through simple observation of the host's physical display. Furthermore, the toggling of screen colour makes it easy to establish whether OmniPanel is issueing the correct commands to the correct host(s), in response to user interaction.
Since much of Omnipanel's complexity resides within the internal system, not all testing can be achieved through real-world observation. Evaluation the behaviour of the resource management functions, for example, requires the inspection of run-time console output to confirm that unused resources are being reclaimed and that internal components are being shared by different widget deployments.

10.2 Function Testing

10.2.1 Widget Deployment

No.	Description	Expected Outcome	Pass?
1.	Test to ensure that selection of a single widget within the library displays the configuration options.	Configuration drop-down becomes visible on control panel.	Yes
2.	Test to ensure that multiple widgets cannot be deployed at once.	Configuration drop-down hides when more than one widget is chosen in the library	Yes
3.	Test to ensure that an unlimited number of hosts can be configured by the user.	New row of input fields is appended to host configuration table when the '+' icon clicked.	Yes
4.	Test to ensure that the user can choose the auto-discover hosts to associate with a widget.	Clicking the 'AutoDiscover Hosts' will display the input for a beacon identifier.	Yes
5.	Test to ensure the host discovery tool functions correctly.	IP addresses of hosts transmitting a user-specified beacon identifer are appended to host configuration table.	Yes
6.	Test to ensure the user specify a range of IP addresses to deploy against.	Clicking the 'Add A Range Of IPs' option displays the input fields for defining a range of IP addresses.	Yes
7.	Test to ensure the the IP range generator functions correctly	Correct IP addresses are generated given a common IP subnet, range-start and range-end values.	Yes
8.	Test to ensure that a widget can be deployed to a specified slot.	The content of a widget appears in the user defined slot of the control page.	Yes

10.2.2 Monitoring State

No.	Description	Expected Outcome	Pass?
1.	Test to ensure widget content automatically updates.	Setup widget to monitor local machine. Changes manually made to a locally monitored machine are propagated to widget content	Yes
2.	Test to ensure the update process pauses during user interaction.	Change to locally monitored machine are not displayed in widget content whilst mouse is hovered over the slot.	Yes
3.	Test to ensure user can alter which remote host they monitor.	When configured against 2 local hosts of differing state, the content of the widget adjust with switching of host.	Yes
4.	Allow user to manipulate widget refresh rate.	Through monitoring of server output, it can be seen that a widget polls less often when a higher refresh rate is chosen	Yes

10.2.3 Manipulating State

No.	Description	Expected Outcome	Pass?
1.	Test to ensure interactions with a single-host deployment are handled correctly.	When the user makes an interaction, the state of a single remote host is adjusted accordingly.	Yes
2.	Test to ensure interactions with a multi-host deployment are handled correctly.	When the user makes an interaction, the state of multiple remote hosts is adjusted accordingly.	Yes
3.	Test to ensure appropriate commands are issued in response to a user interaction.	When using the 'Colour Control' widget, the user-selected colour is displayed on remote display(s)	Yes

10.2.4 Error Handling

No.	Description	Expected Outcome	Pass?
1.	Test to ensure failed SSH connections are handled gracefully.	When an SSH connection cannot be established, the system remains online and sends a report email to deployment owner.	Yes
2.	Test to ensure unimplemented state polling functions are handled gracefully.	When a compulsory function has not been defined within a script, polling ceases and the owner is emailed	Yes
3.	Test to ensure unimplemented application-specific functions are handled gracefully.	When a function specified within a HTML widget does not exist within the associated back-end script, the system remains online and sends an email report to the deployment owner.	Yes

10.2.5 Internal Mechanisms

No.	Description	Expected Outcome	Pass?
1.	Test to ensure the heartbeat mechanism detects when a user has closed their session.	Console output should indicate that a user misses a beat and on the second miss is assumed to have left.	Yes
2.	Test to ensure components are torn-down when a widget is deleted.	Console output should indicate that a StatePoller and CommandChannel objects have died.	Yes
3.	Test to ensure components are torn-down when a users logs out.	Console output should indicate that a StatePoller and CommandChannel objects have died.	Yes
4.	Test to ensure components are torn-down when a users navigates away from the page.	Console output should indicate that a StatePoller and CommandChannel objects have died.	Yes
5.	Test to ensure components are shared whenever possible.	Console output should indicate sharing of a StatePoller object when two deployments are configured against the same remote application.	Yes
6.	Test to ensure that sensitive data is held for no longer than required.	Upon deletion of a widget from a user's control panel, the associated host-file is deleted from the system.	Yes

10.3 Scalability Testing

Testing the true scalability of OmniPanel is a difficult task due to the limited resources available within the lab environment. There is also the question of how to quantify scaleability, thus this section of testing somewhat speculative at best.
To investigate OmniPanel's performance when interacting with multiple remote hosts, a deployment was created to control a group of 10 and then a group of 20 computers in the Computer Science Senior Honours lab. Photographs of this exercise can be found in Appendix E.
The unix command-line tool HTOP [1] was used to monitor the system resources being used by Django framework (upon which OmniPanel runs) and also the SSH connections that are spawned for interaction with remote hosts.
The data gathered using HTOP is presented below:

CPU Usage:

	OmniPanel Core	**SSH**	**Total**
1 Host	0.0	0.0	0.0
10 Hosts	0.0	0.0	0.0
20 Hosts	1.0	0.0	1.0

Memory Usage:

	OmniPanel Core	**SSH**	**Total**
1 Host	1.0	0.1	1.1
10 Hosts	1.0	1.0	2.0
20 Hosts	1.0	2.0	3.0

[1] http://htop.sourceforge.net/

Chapter 11

Evaluation

11.1 Analysis Of Testing

Chapter 10 of this report describes how OmniPanel has been tested in terms of both functionality and scaleability. The results of this testing shall now be discussed.

11.1.1 Function Testing

The results of the function testing shows that the core functionality of OmniPanel works correctly. The state of one or more remote servers is clearly presented within widget content and can be manipulated via interaction with widget input elements. Support for new applications can be successfully introduced to the system through the upload of new front-end and back-end files.
The resource management mechanisms within OmniPanel also seem to function correctly, identifying and tearing-down the components that are no longer needed as well as recognising where an existing component can be used to service multiple requests.

11.1.2 Scalability Testing

The data gathered from scalability testing shows that an SSH connection to a remote host consumes 0.1% of system memory and neglible amount of CPU time.
Adding hosts to the system had no noticeable effect on the memory used by the OmniPanel core and only a very small effect on CPU time (rising from 1% to 2% with the addition of 19 hosts).
The numbers above indicate that the addition of a remote host has next to no noticeable cost to the system. Furthermore, it should be noted that these figures were captured when OmniPanel was deployed on a standard personal use laptop. Consuming 2% of memory corresponds to only 40mb of physical RAM. Deployment on an industrial server, therefore, would add support for many more deployments prior to resource saturation.
Whilst the results of our scalability testing are in no way conclusive, provisional data would suggest that OmniPanel does indeed scale nicely.

11.2 Comparison To Original Objectives

In this section, we will compare the final implementation of OmniPanel against the set of objectives outlined at the beginning of the project:

11.2.1 Primary Objectives

Users can create persistant, personalised Control Pages

A personalised control page can be built by a user through the deployments of widget upon a page. A system-wide library of available widgets is presented to the user which, when selected, displays a configuration menu so they can be associated with remote hosts.
Widgets can be deployed in to one of four slots on the Control Page with the user's deployments being saved against their account for persistant usage.

Users can use their deployed widgets to monitor and manipulate server state

Widgets are defined as HTML documents containing standard HTML input elements and special place-holder strings (encapsulated within '!!' characters). User's can interact with input elements to represent their desired state for the remote server. These interactions are routed to a widget's corresponding back-end script that will generate appropriate commands for issue to remote host(s). The place-holder strings used within HTML definitions are overwritten at run-time by the OmniPanel system to inject current server state prior to presentation to the user.

Users can extend the functionality provided by the OmniPanel system

The functionality of the system can be extended by the user through the upload of new front-end and back-end files. The front-end file is simply the HTML widget definition containing input elements and special template place-holders. The back-end file is a python script containing two compulsory functions used by OmniPanel to retrieve and process server state and one or more widget-specific functions that process a user's interaction with a widget and output appropriate commands.
Once uploaded, new functionality is exposed globally to all users of the system.

11.2.2 Secondary Objectives

A single deployment can monitor and manipulate multiple remote hosts

There is no limit to the number of hosts that can be associated with a widget deployment due to an input table that can be dynamically grown by the user.
Once deployed, user interactions are pushed to all remote hosts automatically. The user can only monitor the state of one host at a time due to limitations in screen space and computational resources. The user can adjust, however, the host that they wish to monitor through interaction with a drop-down list.

Develop a range of front-end and back-end files, demonstrating system flexibility

Currently, two front-end and back-end pairings have been written. The first simply interacts with a remote java application to manipulate and monitor the colour of a

full-screen window. This was developed for use in the testing process as observing the colour of a computer monitor helps us to quickly see whether state is being correctly retrieved and manipulated.
The second front-end and back-end pairing allows the user to interact with a CounterStrike [1] gaming server (See Appendix C). Users are able to monitor and adjust the map currently being played and can also interact with single players on the server (setting them on fire as an admin punishment or kicking them from the server entirely).

11.2.3 Tertiary Objectives

Allow for the modification of existing deployments, adding or removing remote hosts

This objective has not been met due to time restrictions but could easily be implemented. The host auto-discovery tool already demonstrates how a list of IPs can be sent from the OmniPanel server and presented within HTML. Users could then interact with this list to add or remove hosts before submitting the updated list to the server for reprocessing

11.3 Meeting Of Requirements

All functional requirements outlined at the start of this document have been achieved. Non-functional requirements relating to timely system responses and graceful handing of unexpected errors are harder to discuss. The definition of a timely response can vary significantly depending on the user and the nature of the application with which they are interacting. Furthermore, the responsiveness of the system is likely to vary depending on system load. I acknowledge that the relatively small scale testing performed thus far may not give a true representation of timeliness. With regard to the handling of unexpected errors, the system has certainly been written to minimise the number of dangerous assumptions made in the codebase. Whilst it cannot be guaranteed that all edge cases have been covered, I feel confident in saying that the obvious failure modes (SSH login failure, missing back-end functionality) have been dealt with.

[1]http://store.steampowered.com/css

11.4 Comparison With Similar Software

In the Context-Survey earlier in this report, we discuss several technologies that have clear overlap with the OmniPanel project. None of these technologies, however, are trying to solve exactly the same problem, nor do they share the same motivations, as Omnipanel.
OmniPanel, in essence, is a middleware that aims to expose within a web interface, functionality that was originally only available via command-line interaction. Existing projects such as cPanel [3] and Puppet [5], whilst offering web control panels and overviews, work to expose the **configuration** of an application or environment, rather than exposing the functionality of an application itself.
Oracle Salt [7] comes very close to solving the same problem as OmniPanel in the sense that it aims to encapsulate the functionality of what was originally a stand-alone application, and expose it as a distributed system component with which other systems can interact. The key distinction here, however, is that the exposed functionality is for the benefit of system integration, rather than for the benefit of a single end-user. The way in which the component exposes itself is therefore a computer-oriented, web-service style, leveraging the Web Service Definition Language (WSDL) [2]. OmniPanel, in contrast, presents server state and available functionality in a human-readable, point and click HTML interface.
Finally, two distinct properties make OmniPanel unique from any existing projects to-date. Firstly, since manipulation of a remote host is achieved entirely via SSH communication, there is no need to modify the remote host in any way, installing software, opening ports or adjusting security permissions. Systems such as Puppet, however, require a 'Puppet Agent' to be installed on any remote hosts, taking commands and reporting state back to a central 'dashboard' service.
Secondly, all existing Web Control-Panel and server configuration tools are closed-source, offering the user no facility to tailor the system to their needs. OmniPanel, however, is fully extensible making it easy for even the most inexperienced programmers to write new front-end and back-end definitions before uploading them to the system and making the new functionality available to all.

[2]http://www.w3.org/TR/wsdl

11.5 Future Work

Support Deployment Reconfiguration

A tertiary requirement of this project was to provide a mechanism by which an existing deployment could be reconfigured to add or remove hosts, for example. This requirement, unfortunately, has not yet been realised within OmniPanel. Provision of such a facility would no doubt add great value to the project, removing the need to redeploy from scratch whenever the needs of a user change.
Implementing a reconfiguration tool should be possible without too much work. The host auto-discovery mechanism has already demonstrated how AJAX can be used to pass a list of host IP addresses to the client and have them rendered on screen. Displaying the current configuration of a deployment should therefore be a trivial problem. Any amendments made to the list of hosts can then be delivered to OmniPanel via form POST data, as per the original deployment process.

Facilitate Unlimited Deployments

Currently a user can deploy a maximum of four widgets upon their Control Page. This is due to the limited number of widget slots available in their web interface. It would not be too difficult to develop the system so that additional slots can be appended to a user's interface at run-time. This would, however, involve new functionality to be implemented in both the front-end web-browser (which must render the slot on screen) and the internal OmniPanel system (which must be aware of this slot in order to inject content).

Improved Error Reporting

Currently, users are informed of OmniPanel errors via emails directed to their personal accounts. This is far from a perfect solution assuming not only that users have a mail client open when using OmniPanel, but also assuming that emails will be delivered to the user in a timely manner.
A more appropriate reporting mechanism would be to display error messages directly within a user's web control page. Unfortunately, error reporting was not considered when designing OmniPanels architecture and to restructure the system so late in to the project would be foolish.
I would suggest that an error reporting mechanism could when be amalgamated within the existing client heartbeat system. In this revised approach, each heartbeat message could be seen not only as an indicator of a user's presence, but also as their polling for error reports. Any undelivered reports could then be transmitted back to the user in response to their heart-beat message.

Improved Input Validation

Currently, OmniPanel delivers a degree of protection from invalid user input but is not entirely fool-proof. For example, the system responds gracefully to invalid back-end scripts in which compulsory or user-defined functions have not been implemented. The system also protects against some nonsense interactions, for example, a user cannot try to deploy multiple widgets within a single slot as the configuration menu will only display when a single widget is selected in the library.

OmniPanel does not currently perform any validation on the values that are entered in to the host configuration menu itself. For example, a user can enter any value they like for an IP address, rather than enforcing it be a string of numbers and dots. Input validation could be implemented within the OmniPanel core itself or even better, at the client-side using JQuery (thus offloading the expense of validation to the client).

Create User Sign-up Screen

OmniPanel is currently a closed project where new user accounts must be created manually by a system administrator. A user sign-up form must be implemented prior to public release so that user accounts be created autonomously by the OmniPanel core.

Chapter 12

Conclusion

12.1 Achievements

12.1.1 Flexibility

OmniPanel has been developed in such a way that a web control interface can be developed for *any* application that originally exposes its functionality via command line.
OmniPanel is, therefore, a hugely flexible system that can be used to monitor and interact not only with server applications, but also with local applications such as media players or cronjob managers [1].

12.1.2 Extensability

Having a system that is flexible enough to interact with any command-line application is useless without also making it possible for newly developed functionality to be introduced.
OmniPanel promotes extensability through the provision of it's widget upload facility, whereby new widgets and back-end script can be uploaded and made available for use immediately.
For scenarios where front-end and back-end implementations do not exist for your application, anyone with a basic grasp of HTML and python scripting can implement the new functionality themselves. This is made possible by the sensible manner in which OmniPanel routes data between front-end and back-end files whilst also ensuring internal complex behaviors are hidden from the user entirely.

12.1.3 Scaleability

Provisional studies of OmniPanels performance with multi-host deployments suggest that OmniPanel copes well when required to interact with many machines simultaneously. This can be attributed to the careful resource management implemented within OmniPanel, tearing-down superfluous components whenever possible and detecting scenarious where existing components can be re-used.

[1]http://unixhelp.ed.ac.uk/CGI/man-cgi?crontab+5

12.2 Final Conclusion

OmniPanel achieves all the primary and secondary objectives that were outlined at the beginning of the project. Whilst there are several ways in which the project could be improved, the desired core functionlity has certainly been attained.
OmniPanel builds upon work from other web-based control services to expose application level functionality rather than exposing system configuration only. OmniPanel succeeds in presenting its functionality in a friendly, uncomplicated manner. Users of the system should have no problem interacting with familiar HTML forms to manipulate that state of their remote applications. Furthermore, developers with even a rudimentary knowledge of HTML and python should be write new front-end and back-end files so that support for new applications appears all the time. The use of strong software engineering practices such as revision control, code documentation and an agile approach to programming have enabled me to deliver a product on time and to a high standard.

Chapter 13

Appendices

A Java Colour Server Application

The source code for the Java application was used for function and scaleability testing can be found in the directory Code/tools/ColourServer
The application can be run using the following command:

```
java ColourServer
```

B Colour Control Widget And Script

The widget definition for the Colour Control functionality can be found in the directory Code/frontends/ColourWidget.html
The back-end script for the Colour Control functionality can be found in the directory Code/backends/colour.py

C CounterStrike Widget And Script

The widget definition for the CounterStrike functionality can be found in the directory Code/frontends/CounterStrike.html
The back-end script for the CounterStrike functionality can be found in the directory Code/backends/counterstrike.py

D Host Auto-discovery Beacon Agent

The python Beacon Agent script can be found in the directory Code/tools/Beacon
The application can be run using the following command:

```
python Beacon.py
```

E Pictures From Scaleability Testing

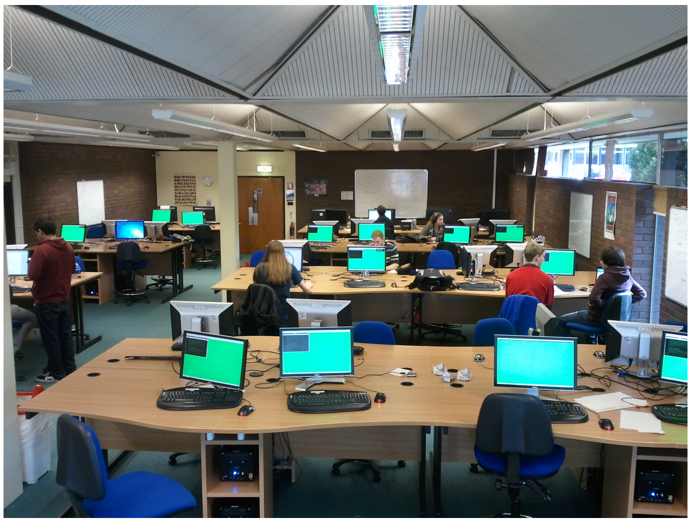


Figure 13.1: Using OmniPanel to control the Java ColourServer application on 20 remote hosts

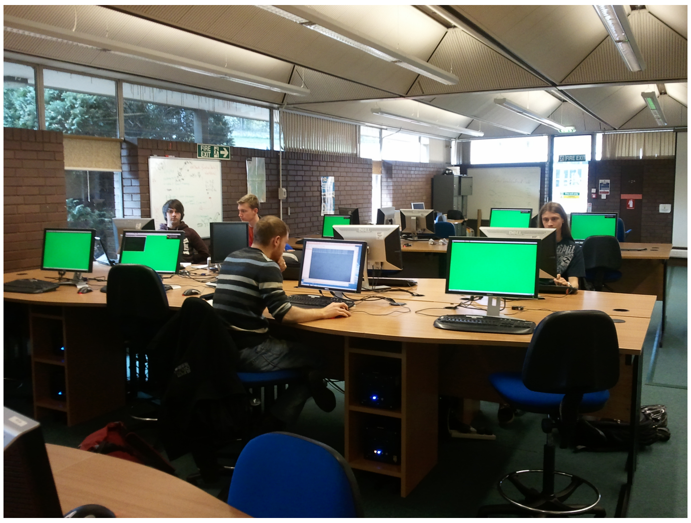


Figure 13.2: Using OmniPanel to control the Java ColourServer application on 20 remote hosts

Bibliography

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[2] M. Bakery and R. Buyyaz. Cluster computing at a glance. *High Performance Cluster Computing: Architectures and Systems*, pages 3–47, 1999.

[3] cPanel. cpanel and whm overview. `http://www.cpanel.net/products/cpanelwhm/`, 2011.

[4] Thomas Goirand. Domain technologie control. `http://gplhost.com/software-dtc.html`, 2010.

[5] Puppet Labs. What is puppet. `http://puppetlabs.com/puppet/what-is-puppet/`, 2012.

[6] J.P. Navarro, R. Evard, D. Nurmi, and N. Desai. Scalable cluster administration-chiba city i approach and lessons learned. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 215–221. IEEE, 2002.

[7] Oracle. Oracle salt overview. `http://docs.oracle.com/cd/E13161_01/salt/docs10gr3/overview/over.html`, 2008.

[8] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. Npaci rocks: tools and techniques for easily deploying manageable linux clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):707–725, 2003.

[9] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14 –22, sept.-oct. 2009.

[10] Yi Yu and S. Bhatti. Energy measurement for the cloud. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, pages 619 –624, sept. 2010.