Science & Technology Facilities Council

Technical Report

Developing a Resilient Power Management Portal for an Enterprise Infrastructure

Adam Horwich

June 2010

RAL-TR-2010-018

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

Developing a Resilient Power Management Portal for an Enterprise Infrastructure

Abstract

The Scientific Computing Technology (SCT) group within the Science and Technology Facilities Council's (STFC) e-Science centre is responsible for managing large scale cluster computing services for the UK's e-Infrastructure as well as STFC's core Facilities. Such an infrastructure needs managing, and SCT has invested a great deal of time and effort in finding suitable products to streamline the System Administration duties where possible, and develop in-house tools when solutions either do not exist or do not justify the cost. Alongside requirements to provide power management functionality to an OS commissioning service, SCT also saw the need to extend functionality to cover all power outlets available in the SCT Infrastructure so a consistent view could be maintained. To meet this challenge, a new service was developed which manages Power Distribution Units (PDU), Intelligent Platform Management Interfaces (IPMI), and VMware Virtual Machines. With this service, authenticated System Administrators can perform fine grained power control of servers and Virtual Machines with information collected from both Oracle and MySQL databases. It employs a database caching technique and runs in a fault tolerant environment to provide a resilient service in the event of building power failure; a scenario where the service is deemed most needed. It gives a consistent view of an enterprise infrastructure, with fine grained control for power management. For the first time, SCT can perform fully automated Operating System deployment on virtually any host in its infrastructure.

Table of Contents

1. Introduction

The Scientific Computing Technology (SCT) group [1] within the Science and Technology Facilities Council's (STFC) e-Science centre is responsible for managing large scale cluster computing services for the UK's e-Infrastructure as well as STFC's core Facilities. Such an infrastructure needs managing, and SCT has invested a great deal of time and effort in finding suitable products to streamline the System Administration duties where possible, and develop in-house tools when solutions either do not exist or do not justify the cost. Alongside requirements to provide power management functionality to an OS commissioning service, SCT also saw the need to extend functionality to cover all power outlets available in the SCT Infrastructure so a consistent view could be maintained. This technical report outlines the development of a new tool, the SCT PDU & IPMI Management Portal (PIMP), designed to address the requirements necessary in providing an enterprise power management service, and how best practices for naming conventions emerged as a result.

Requirements

Before development commenced on a service to manage and monitor PDU power status in the SCT computing infrastructure it was important to define the precise requirements of the service and how it would interact with other services. The following list of requirements was generated after consultation with SCT Service and System Administrators:

- Web portal interface with remote command line interface for Kickstart [2] service
- Modular and expandable to accommodate new racks and clusters
- Failsafe and resilient
- Secure, authenticated access
- Interact with the SCT Oracle Cable Database and MySQL LAN Inventory Database
- Operate on PDUs, IPMI devices and Virtual Machines (VMs)
- AJAX PHP design

2. Interface

Interface Prototyping

Having elected on designing the interface in PHP due to its friendly development environment and support for Simple Network Management Protocol (SNMP) [3], Oracle, and MySQL interaction, the next step was to decide how the interface would need to work. Firstly, it became apparent that there is a large volume of information to describe the SCT Infrastructure and, as such, an asynchronous mechanism of providing only relevant information was essential. To this end, investigations were made into existing design frameworks for tabbed interfaces. The tabbed interface approach works well because each tab can represent a cluster, which would then load only the related racks and PDUs. A series of solutions were researched and tested, however, many were developed by individuals who no longer offer support or maintenance for the code. Through trial and error working with existing designs, investigations were made into jQuery [4] which is a JavaScript API, giving a programmatic approach to developing websites, though a developed and maintained library. Alongside jQuery there is jQuery UI [5] which can be used to develop pre-defined user interface objects rapidly and with minimal code. This led to a very rapid development of the GUI shown in Figure 2.1.

Interface Design

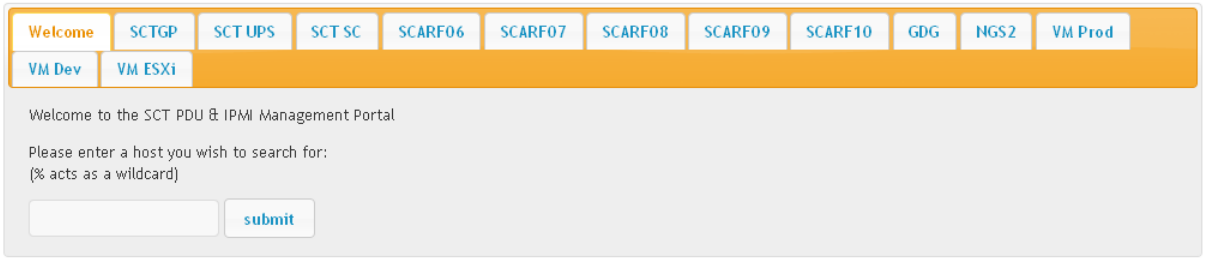

Figure 2.1

Here in Figure 2.1 we see a basic tabbed interface, implemented by jQuery UI, accommodating all of the 'cluster' definitions in SCT's Infrastructure. Each tab will dynamically call a PHP script which populates the tab pane with appropriate tables of information; the code can be seen in Figure 2.2.

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <title><? echo $title;?></title>
    <link rel="stylesheet" type="text/css" href="css/power.css"/>
    <link rel="stylesheet" type="text/css" href="css/ui-lightness/jquery-ui-1.8.custom.css"/>
    <script type="text/javascript" src="scripts/jquery-1.4.2.min.js"></script>
    <script type="text/javascript" src="scripts/jquery-ui-1.8.1.custom.min.js"></script>
    <script type="text/javascript" src="scripts/functions.js"></script>
  </head>
<body>
<div class="power">
  <div id="tabs">
    <ul>
        <li><a href="display.php">Welcome</a></li>
        <li><a href="display.php?cluster=SCTGP"><span>SCTGP</span></a></li>
        <li><a href="display.php?cluster=SCTUPS"><span>SCT UPS</span></a></li>
        <li><a href="display.php?cluster=SCTSC"><span>SCT SC</span></a></li>
        <li><a href="display.php?cluster=SCARF06"><span>SCARF06</span></a></li>
        <li><a href="display.php?cluster=SCARF07"><span>SCARF07</span></a></li>
        <li><a href="display.php?cluster=SCARF08"><span>SCARF08</span></a></li>
        <li><a href="display.php?cluster=SCARF09"><span>SCARF09</span></a></li>
        <li><a href="display.php?cluster=SCARF10"><span>SCARF10</span></a></li>
        <li><a href="display.php?cluster=GDG"><span>GDG</span></a></li>
        <li><a href="display.php?cluster=NGS2"><span>NGS2</span></a></li>
        <li><a href="virtual.php?cluster=PRO"><span>VM Prod</span></a></li>
        <li><a href="virtual.php?cluster=DEV"><span>VM Dev</span></a></li>
```

```
        <li><a href="virtual.php?cluster=ESXI"><span>VM ESXi</span></a></li>
    </ul>
  </div>
</div>
<div id="dialog"></div>

</body>
</html>
```

Figure 2.2

The code in Figure 2.2 essentially comprises of the jQuery and jQuery UI libraries, a small amount of CSS customisation, a jQuery UI theme, and a custom JavaScript file which provides asynchronous functions for display.php. The parameter passed to display.php in the code above will ensure that only that set of racks will be displayed, with the default being no parameter for the welcome page. Figure 2.3 shows the result of clicking the SCTGP tab.

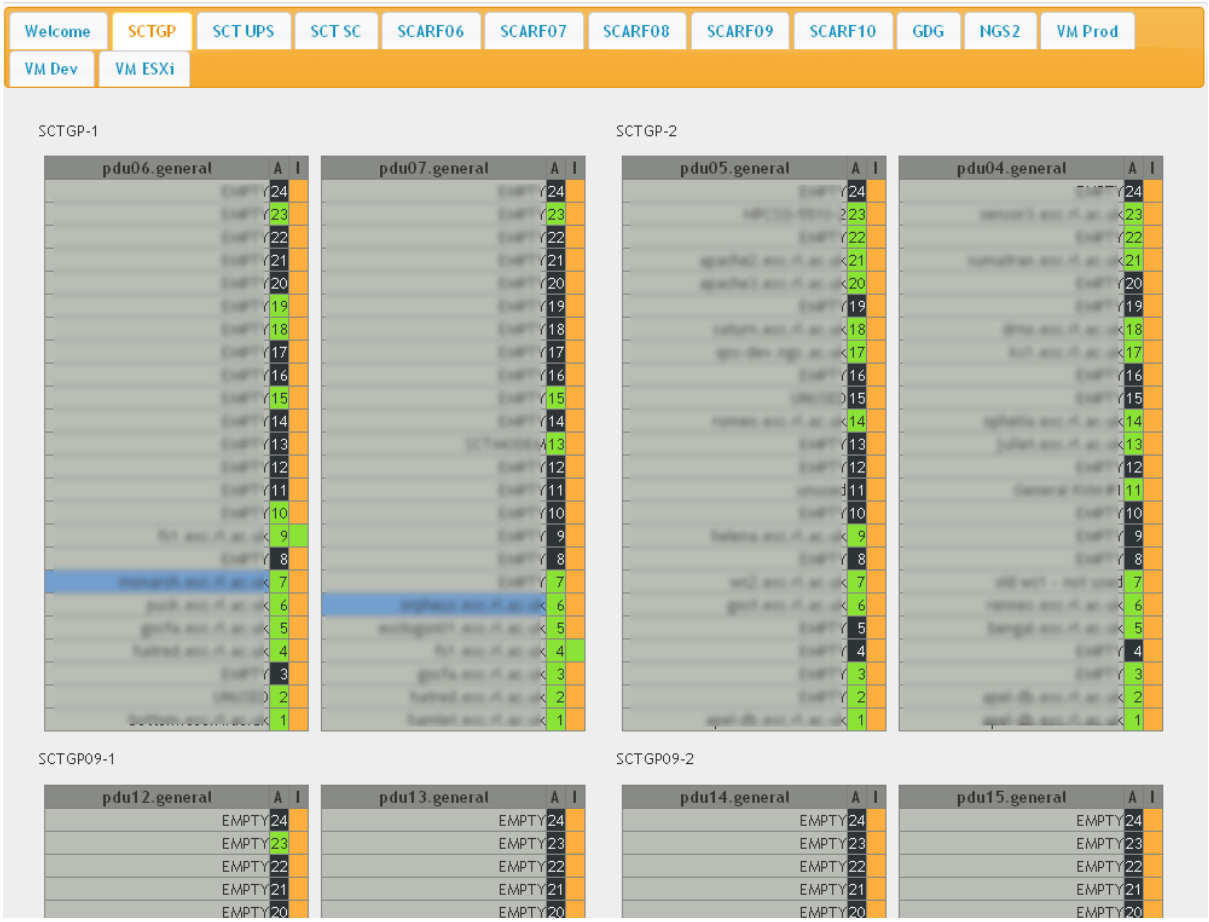

Figure 2.3

In Figure 2.3 we can see how racks are divided into sections containing the discovered number of PDUs. Within this is a list of all the devices connected to PDU ports and the power state of the PDU port itself in column 'A'. Column 'I' shows the status of the IPMI [6] interface; here orange represents an unsupported host, green that the IPMI device is reporting the power to be on, and red would demonstrate that the query timed out. Similarly between both columns black is used to signify that the device reports the power to be off. Where a hostname contains a blue background, this signifies that the host is responsible for more than one server (i.e. a VMware ESX Host [7]) or that the same PDU port is connected to more than one host. Whenever a coloured box is clicked, to perform an operation on its associated host, a jQuery UI Dialog box appears, giving the user a series of options they can perform if they have been successfully authenticated. Unlike the tab pane data, clicking on a host power box will present the user with live information about a host.

3. Presenting Data

The majority of the frontend is generated from display.php which queries the data sources to generate tables displaying PDU power status information, and IPMI where available. In the event that no '$cluster' variable is defined, as shown in Figure 2.2, the welcome screen is presented, seen in Figure 2.1. Otherwise the cluster variable passed from the tabbed interface is used as a parameter for the getPDUs function.

Naming Conventions

In order for the process of discovering PDUs and Racks within a cluster to operate successfully, it became apparent that a standardised naming convention was essential to minimise the complexities involved in displaying data. To this end a set of recommendations were developed to ensure that existing equipment could be renamed to meet the criteria, and any future kit would have a sensible naming convention:

- PDUS - pdu[xx].[service type] (e.g. pdu07.general pdu04.scarf)
- CLUSTERS - Unique name (e.g. SCARF07 SCARF08)
- RACKS - Cluster name suffixed by a digit (e.g. SCARF07-1 SCTGP09-2)
- IPMI - IPMI-[hostname] (e.g. IPMI-hatred.esc.rl.ac.uk)
- ESX hosts need to be defined in service configuration file
- ESX hosts need a user agent created to make power requests

Displaying Clusters

Figure 3.1 below shows the 'getPDUs' function which will return a dictionary using the name of the rack as a key and its contained PDUs as an array value. An example of this is shown in Figure 3.2.

```
function getPDUs($cluster){
  $result = query("oraclerack",array("rack_name","rack_id"),"rack_name","%$cluster%");
  foreach ($result as $row) {
      $result2 = query("oraclecomp","component_name","rack_id",$row[1]);
      foreach($result2 as $row2) {
        if(substr($row2,0,3) == 'pdu'){
          $rack[] = $row2;
        }
      }
      if (isset($rack)) {
        $pdus["$row[0]"] = $rack;
      } else {
      }
      unset($rack);
  }
  return $pdus;
}
```

Figure 3.1

We can see from this code that the flat file database is queried using the query() function (see **Appendix A**) to find the rack_ids and rack_names for racks which match the pattern passed into the function. The % signs in the query are used to signify that a partial match is required. Once a series of rack names have been produced, these are iterated to find the components contained, and where a component contains pdu in their name, the information is recorded in an array.

```
Array
(
    [SCTGP-1] => Array
        (
            [0] => pdu06.general
            [1] => pdu07.general
        )
    [SCTGP-2] => Array
        (
            [0] => pdu05.general
            [1] => pdu04.general
        )
    [SCTGP09-1] => Array
```

```
        (
            [0] => pdu12.general
            [1] => pdu13.general
        )
    [SCTGP09-2] => Array
        (
            [0] => pdu14.general
            [1] => pdu15.general
        )
)
```

Figure 3.2

The PHP file 'display.php' will subsequently iterate per rack and then per PDU to group together logical information in tables.

```
$racks = getPDUs($cluster);
$labels = array_keys($racks);
$j=0;
$configfile = '/etc/cableguy.conf' ;
$iniarray = parse_ini_file($configfile)
     or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;
$vmhosts = $iniarray["vmdev"] . " " . $iniarray["vmprod"] . " " . $iniarray["vmesxi"];
$esxhosts = array_map('trim', explode(" ",$vmhosts));
echo "<div class=\"hack\">";
foreach ($racks as $rack) {
  echo "<div class=\"rack\">\n";
  echo "<p>$labels[$j]</p>\n";
  foreach ($rack as $pdu) {
    echo "<div class=\"pdudiv\">\n";
    echo "<table class=\"pdu\">\n";
    echo "<tr><th>$pdu</th><th>A</th><th>I</th></tr>\n";
```

Figure 3.3

For each table generated, there are then two key functions executed, which provide the PDU power status (readPDU) of every port it contains, and a list of all the hosts which are connected to that PDU (getAllHosts). Please see **Appendix B** and **Appendix C** for further details on these functions. The getAllHosts function is critical here as it can tell us if more than one host is connected to a given PDU port. With this we can highlight these ports to ensure the System Administrator is aware that they may potentially be powering off more than one host. The last two lines of Figure 3.3 form the table and table header component of a PDU. Everything else in that table is dynamically generated from the results of the two aforementioned functions.

```
for ($i = count($results); $i > 0; $i--) {
  if (count($hostnames[$i]) > 1) {
    $multi="multi";
  } elseif (in_array($hostnames[$i][0],$esxhosts)) {
    $multi="multi";
  } else{
    $multi="normal";
  }
  foreach($hostnames[$i] as $connectedhost) {
    echo "<tr>";
    echo "<td class=\"$multi\">$connectedhost</td><td class=\"$results[$i]\" ";
    if ($results[$i] != "unsupported") {
      echo "style=\"cursor: pointer;\" onClick=\"showPDU('port=$i&pdu=$pdu','$connectedhost'); return
false;\" >$i</td>\n";
    } else {
      echo ">$i</td>";
    }
    $ipmistatus = getIPMIStatus($connectedhost);
    echo "<td class=\"$ipmistatus\" ";
    if ($ipmistatus != "unsupported") {
      echo "style=\"cursor: pointer;\" onClick=\"showIPMI('hostname=$connectedhost','$connectedhost');
return false;\"></td>\n";
    } else {
      echo "></td>\n";
    }
    echo "</tr>\n";
  }
  unset($multi);
}
echo "</table></div>\n";
```

Figure 3.4

'$hostnames' is the returned multidimensional array from getAllHosts and '$i' represents the 'for' loop which encompasses it, based on the number of PDU ports on the PDU we are tabulating ('$results'). For each hostname we find connected to that numbered PDU port (and it's important to note we iterate backwards through the array as PDU ports are numbered from bottom to top) we draw a table row and if there are multiple, we make sure they use the 'multi' CSS class which shades the background colour in blue. For each host entry we make one or two boxes clickable depending whether the PDU supports per port controls, and whether an IPMI device is configured (determined by getIPMIStatus **Appendix D**).

Displaying Power Controls

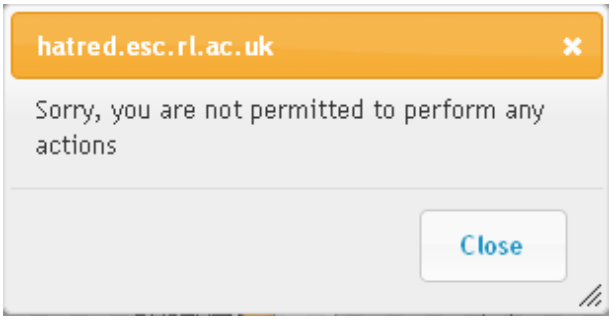

Figure 3.5

Figure 3.6

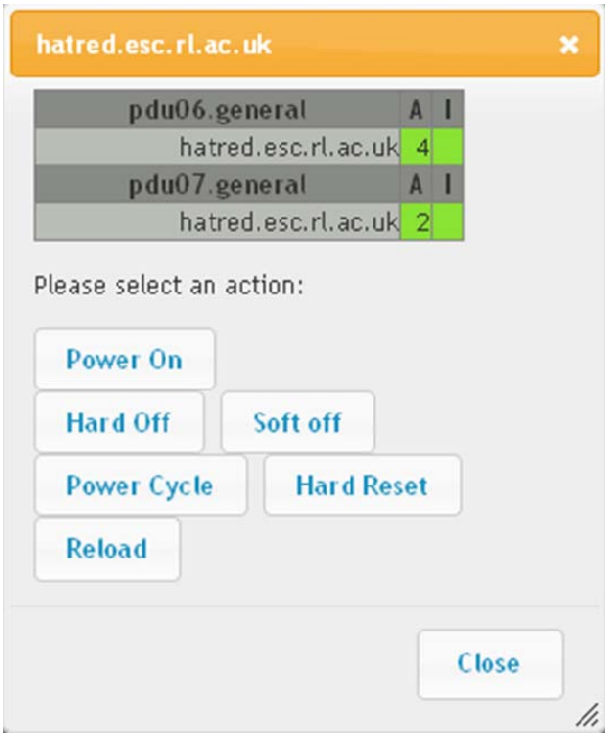

Figure 3.7

If a user does not have permission to perform any power operation, when they click an A or I box, they will receive a dialog box as shown in Figure 3.5. If an authenticated user clicks an A box, they will receive a dialog box containing something similar to Figure 3.6, and if they click to alter the IPMI status, they will receive a dialog box as shown in Figure 3.7. The motivation for not even showing live host status for unauthenticated users is such that a malicious user may put excess stress on the databases and physical components by repeatedly selecting hosts. The unauthorised dialog box contains no host processing. As shown in the code from Figure 3.4, these dialog boxes are called via a JavaScript function, either 'showPDU' or 'showIPMI'.

Access Control

Before a user can gain access to power controls, they must first be vetted against a white-list of approved X.509 Certificate Distinguished Name (DN) [8] strings signed by the UK e-Science Certificate Authority [9].

```
$admin = (isset($_SERVER["SSL_CLIENT_S_DN"])) ? $_SERVER["SSL_CLIENT_S_DN"] : "Unknown";
$dnlist = '/etc/dn-list';
$lines = file($dnlist);
$authorized = FALSE;
for($i=0;$i<count($lines);$i++) {
  if ($admin == trim($lines[$i])) {
      $authorized = "TRUE";
  }
}
```

Figure 3.8

These lines of code are included in each of the PHP files used to control PDU, IPMI and VM operations. The user's DN is extracted from their certificate, which they need to possess in order to reach the service, and then compared against a list of approved DN strings, if a match is found then the user has permission to perform operations on the selected host/PDU port.

showPDU

The only parameters passed to this JavaScript function are the PDU port number and PDU name clicked by the user. This cuts down on the number of assumptions the service can make about the user's actions, and mitigates situations where multiple hostnames may exist in the service.

```
function showPDU(str,strtitle)
{

if(!strtitle) strtitle='Error message';

$('#dialog').dialog('destroy');
$('#dialog').show();
$('#dialog').html('<center><img src=\'img/ajax_preloader.gif\'/ border="0"/></center>');
$('#dialog').load('commands/pdu.php?' + str);
$('#dialog').dialog({
        resizable: true,
        modal: true,
        buttons: {
                Close: function() {
                        $(this).dialog('close');
                }
        },
        overlay: {
        backgroundColor: '#000',
        opacity: 0.9
        },
        title:strtitle
});
}
```

Figure 3.9

The function shown in Figure 3.9 shows some jQuery syntax for creating a jQuery UI Dialog box populated by executing a pdu.php script and passing the PDU port and PDU name as parameters.

showIPMI

This operation works slightly differently to showPDU because IPMI devices are 'host' specific. The only parameter which needs passing for this function is the hostname. This value is then passed onto ipmi.php which, if it finds a match in the hostname inventory, will process status output accordingly.

pdu.php, ipmi.php, vm.php

On successful authentication, the first thing to occur, programmatically, is a switch statement based on whether a 'status' parameter has been passed in the URL. This will operate on a PDU, IPMI, or VM before displaying its current status. Typically when a user first clicks a host from the display.php page, no status information will have been passed so this will be skipped.

```
if (isset($_REQUEST["status"])) {
  switch($_REQUEST["status"]) {
     case "on":
       $reply = setPDU($pdu, $port, "on");
       break;
     case "off":
       $reply = setPDU($pdu, $port, "off");
       break;
     case "onall":
       foreach($hosts as $hostname) {
         if ((strtoupper($hostname) != 'EMPTY') && (strtoupper($hostname) != 'UNUSED')) {
           $ports = getPortStatus($hostname);
           foreach($ports as $portinfo) {
             $reply = setPDU($portinfo[0], $portinfo[1], "on");
           }
         } else {
           echo "You cannot perform multiple operations on EMPTY ports\n";
         }
       }
       break;
     case "offall":
       foreach($hosts as $hostname) {
```

```
        if ((strtoupper($hostname) != 'EMPTY') && (strtoupper($hostname) != 'UNUSED')) {
          $ports = getPortStatus($hostname);
          foreach($ports as $portinfo) {
            $reply = setPDU($portinfo[0], $portinfo[1], "off");
          }
        } else {
          echo "You cannot perform multiple operations on EMPTY ports\n";
        }
      }
      break;
    default:
      echo "Invalid PDU operation specified\n";
      break;
  }
}
```

Figure 3.10

It's important to note that a further DN string check is made when the setPDU/setIPMIstatus/setVM function is called (**Appendix E**). After the switch statement, the dialog box begins to render a table displaying the number of hosts affected by the box which was clicked in the tabbed interface (i.e. if a PDU port was clicked, there may be more than one device connected, or the connected device may have more than one power supply). Next the code splits into two, depending on whether an 'empty' PDU port was clicked or not. The objective here is to not waste unnecessary time querying devices and databases to return information on an unpopulated port.

```
if ((isset($hostname)) && (strtoupper($hostname) != 'EMPTY') && (strtoupper($hostname) != 'UNUSED')) {
  $ports = getPortStatus($hostname);
  echo "<table class=\"pdu\">\n";
  while ($portinfo = array_shift($ports)) {
    $pducount++;
    $ipmistatus = getIPMIStatus($portinfo[2]);
    echo "<tr>\n";
    echo "<th>$portinfo[0]</th><th>A</th><th>I</th>\n";
    echo "</tr>\n";
    echo "<tr>";
    echo "<td>$portinfo[2]</td><td class=\"$portinfo[3]\">$portinfo[1]</td><td class=\"$ipmistatus\">
</td>";
    echo "</tr>\n";
  }
  echo "</table>\n";
} else {
  $pdustatus = readPDUPort($pdu,$port);
  #$ipmistatus = getIPMIStatus($hostname);
  echo "<table class=\"pdu\">\n";
  echo "<tr>\n";
  echo "<th>$pdu</th><th>A</th><th>I</th>\n";
  echo "</tr>\n";
  echo "<tr>";
  echo "<td>$hostname</td><td class=\"$pdustatus\">$port</td><td class=\"unsupported\"></td>";
  echo "</table>\n";
}
```

Figure 3.11

Finally, a series of buttons are displayed depending on various circumstances, to provide the user with a set of controls.

```
if (strtoupper($hosts[0]) == "EMPTY" || strtoupper($hosts[0] )== "UNUSED") {
  echo "<p>You are intending to modify the power status of an unoccupied or undocumented port</p>\n";
}
echo "<p>Please select an action:</p>\n";
if ($pducount > 1) {
  if(count($hosts) > 1) {
    echo "<p>(this PDU port is powering multiple hosts)</p>\n";
  } else {
    echo "<p>(this host is connected to several pdu ports)</p>\n";
  }
  echo "<button class=\"fg-button ui-state-default ui-corner-all\"
onClick=\"reloadPDU('port=$port&pdu=$pdu&status=onall');\">Power All on</button>\n";
  echo "<button class=\"fg-button ui-state-default ui-corner-all\"
onClick=\"reloadPDU('port=$port&pdu=$pdu&status=offall');\">Power All off</button><br>\n";
} else {
  echo "<button class=\"fg-button ui-state-default ui-corner-all\"
onClick=\"reloadPDU('port=$port&pdu=$pdu&status=on');\">Power On</button>\n";
```

```
  echo "<button class=\"fg-button ui-state-default ui-corner-all\"
onClick=\"reloadPDU('port=$port&pdu=$pdu&status=off');\">Power Off</button><br>\n";
}
  echo "<button class=\"fg-button ui-state-default ui-corner-all\"
onClick=\"reloadPDU('port=$port&pdu=$pdu');\">Reload</button>\n";
```

Figure 3.12

There are a few minor differences in ipmi.php and vm.php, but they mainly revolve around how the information is gathered. The intention of the pop-up dialog box is that it will show live information compared to what may potentially be static or outdated information on the tabbed display area. The buttons call reload based JavaScript functions, so as to give the appearance that it has seamlessly updated the dialog box when an action is performed. This is due to the fact that any 'status' operation is performed before the rest of the page is rendered, so the eventual status which is displayed, will be based on the action performed by the user.

4. Virtual Machines

The process of communicating with Virtual Machines and VMware ESX hosts is slightly different to that of IPMI and PDU devices, as a separate API [10] needs to be used, as does the method for polling power status. In terms of displaying cluster information, a separate 'virtual.php' script is used instead of the more standard 'display.php'.

```
<?php
  $cluster = (isset($_REQUEST["cluster"])) ? $_REQUEST["cluster"] : "";
  include("includes/functions.php");
    $configfile = '/etc/cableguy.conf' ;
    $iniarray = parse_ini_file($configfile)
            or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;

  if ($cluster == "DEV") {
    $cluster = array_map('trim', explode(" ",$iniarray["vmdev"]));
  } elseif ($cluster == "PRO") {
    $cluster = array_map('trim', explode(" ",$iniarray["vmprod"]));
  } elseif ($cluster == "ESXI") {
    $cluster = array_map('trim', explode(" ",$iniarray["vmesxi"]));
  }
    echo "<div class=\"hack\">";
    foreach($cluster as $host) {
      echo "<div class=\"rack\">\n";
      #echo "<div class=\"pdudiv\">\n";
      echo "<table class=\"pdu\">\n";
      echo "<tr><th>$host</th><th>A</th></tr>\n";
      $result = readVMs($host);
      foreach ($result as $vm) {
        $state = readVMstatus($vm[1],$host);
        echo "<tr>";
        echo "<td class=\"normal\">$vm[0]</td><td class=\"$state\" style=\"cursor: pointer;\"
onClick=\"showVM('host=$host&guest=$vm[0]','$vm[0]'); return false;\"></td>\n";
        echo "</tr>\n";
      }
      echo "</table></div>\n";
    }
?>
```

Figure 4.1

Clusters here are defined in the service configuration file, as VMware do not provide a mechanism of querying a vCenter Server to list the hosts and how they are clustered. Additionally, we also want to provide power options for ESXi hosts which are neither managed nor licensed, so these must be defined statically. The output is the same, except that VMware specific functions are called to find connected Virtual Machines and the respective power status, readVMs and readVMstatus (**Appendix G**). Also, because the processing of these commands is very slow using the VMware provided remote command line interface, the operation which caches the databases also caches VM power status into a table. So results displayed by clicking a tab are up to two hours old. To mitigate this, clicking on a Virtual Machine will poll that individual host to query its current power status, which is subsequently fed back into the cache table to update the offline record. A current limitation of the service is that it only provides methods for querying and performing actions on VMware virtualisation solutions.

5. Searching

As SCT's infrastructure grows and expands, the requirement to rapidly know which tab a host may be contained in grows also. In anticipation of this a search feature was incorporated into the welcome page. This lightweight code (shown in Figure 5.1) compares a validated partial or fully qualified hostname against the hostname database to find where it is located in the SCT infrastructure.

```
<?php
include("includes/functions.php");
$string = (isset($_REQUEST["search"])) ? $_REQUEST["search"] : "";
if (isset($_REQUEST["search"])) {
  $search = preg_replace("/[^a-z0-9\\-\\.\\%]/i","",$string);
  if(strlen($search) > 4) {
    $results = findHostRack($search);
    if ((isset($results)) && (count($results) > 0)) {
      echo "<table class=\"pdu\"><tr><th>Hostname</th><th>Rack</th></tr>\n";
      foreach($results as $row) {
        echo "<tr><td>$row[0]</td><td>$row[1]</td></tr>";
      }
      echo "</table>\n";
    } else {
        echo "No results found\n";
    }
  } else {
    echo "Your search criteria must be 5 characters or longer\n";
  }
}

?>
```

Figure 5.1

As well as validation in the php file, there are also some additional checks performed in the JavaScript which handles the input text box and submits the request to search.php seen in Figure 5.2.

```
function findPage() {
  var str=document.getElementsByName("racksearch")[0].value.split(' ').join('');
  if(str.length > 4) {
    $('#searchresult').show();
    $('#searchresult').html('<center><img src=\'img/ajax_preloader.gif\'/ border="0"/></center>');
    $('#searchresult').load('search.php?search=' + str);
  } else {
    $('#searchresult').html('Your search criteria must be 5 characters or longer');
  }
}
```

Figure 5.2

The above code ensures that searches are more than 4 characters long and all spaces are removed. To provide an asynchronous interaction, the results of search.php are pushed into an empty div generated below the search box. Validation by search.php strips out all characters which are not alphanumerical, dashes, full stops or percentage signs. If the string passes our validation test it is then passed to findHostRack (**Appendix F**) which returns all matching hostnames to the query string and their connected rack. The results are then placed in a simple table and displayed for the user.

6. Caching Data

The code shown in Figure 6.1 outlines the process of storing data selected from a component table. The caches are generated in the following fashion.

1. A path, configured in /etc/cableguy.conf, is allocated to a table
2. A temporary file is opened
3. A query is performed to select all rows
4. The cache file is opened and column headers are written as the first line separated by '|||'
5. Each row is parsed and written into the file below its respective header
6. The file is closed and the cache file gets overwritten by the temp one

```
$stid = oci_parse($oracle,"SELECT component_id, component_name, comp_type_id, rack_id FROM
ngsdb0047.component");
$r = oci_execute($stid);
$ocomptmp = $oraclecomp . ".tmpx";
$fh = fopen($ocomptmp, 'w');
fwrite($fh, "component_id|||component_name|||comp_type_id|||rack_id\n");
while ($int = oci_fetch_row($stid)) {
  $string = $int[0] . "|||" . $int[1] . "|||" . $int[2] . "|||" . $int[3] . "\n";
  fwrite($fh, $string);
}
fclose($fh);
$result = exec("chown apache $ocomptmp; mv -f $ocomptmp $oraclecomp");
```

Figure 6.1

Caching VM information is slightly more complex as we need prior knowledge of all the ESX and ESXi hosts we have and on each of those hosts there needs to be a local account which can read (and write if necessary) state data. Querying an ESX host will generate a list of all VMs contained on that server, plus a reference to their configuration file. Each of these returned VMs then need to be queried to check their power status. The whole process can take up to 5 minutes for 80 VMs over 9 ESX and ESXi hosts. The formatted flat file table is the same as those generated by querying databases, to ensure compatibility.

7. Remote Interface

The project was set out with a requirement to provide an external interface that SCT could use for Kickstart deployment of Operating Systems. Using the same pdu.php, ipmi.php and vm.php and an additional query.php file, information can be extracted about a given hostname supplied remotely via CURL from the SCT Kickstart server. For sanity checking, the first thing a remote Kickstart service should do is 'query.php' a hostname to determine how it is connected and what is controllable.

```
<?php
  include("../includes/functions.php");
  $ks2host = (isset($_REQUEST["ks"])) ? $_REQUEST["ks"] : null;
  $admin = (isset($_SERVER["SSL_CLIENT_S_DN"])) ? $_SERVER["SSL_CLIENT_S_DN"] : "Unknown";
  if (isset($_SERVER['REMOTE_ADDR'])) {
    $remotehost = trim(gethostbyaddr($_SERVER['REMOTE_ADDR']));
  }
  if ((isset($remotehost)) && ($remotehost == "ks2.esc.rl.ac.uk") && (isset($ks2host)) ) {

    $ports = getPortStatus($ks2host);
    $ipmistatus = getIPMIStatus($ks2host);
    echo "Host report: $ks2host\n";
    if ($ipmistatus != "unsupported") {
      echo "IPMIstatus: $ipmistatus\n";
    }
    foreach ($ports as $port) {
      echo "PDU: $port[0] PORT: $port[1] STATUS: $port[3] \n";
      $result = getConnectedHost($port[0],$port[1]);
      foreach($result as $row){
        if($row != $ks2host) {
          echo "This PDU also powers: $row\n";
        }
      }
    }
    $esx = getHostFromGuest($ks2host);
    if (isset($esx)) {
      $vmx = getVMpath($ks2host,$esx);
      $status = readLiveStatus($vmx, $esx);
      echo "VMhost: $esx VMstatus: $status\n";
    }
  } else {
    echo "Usage: -d ks=[hostname]\n";
  }
?>
```

Figure 7.1

Figure 7.1 shows the 'query.php' function that, like all others in the 'command' umbrella, requires certificate authentication or the connecting hostname to be configured by the service (i.e. the SCT Kickstart service hostname). Upon successful authentication, an attempt will first be made to see if the hostname is connected to a PDU port, then secondly if it has IPMI configured. As a host may possess more than one power supply, this function can return all of the PDUs connected and their respective ports. Finally, a check will be made to see if the hostname is a virtual machine. In this event, live information on VM state will be returned. From this set of information a remote service should be able to determine programmatically which of the ipmi/pdu/vm scripts needs to be executed to power cycle the host as part of the commissioning process.

8. Conclusion

Encompassing all the features outlined in this technical report it can be seen that, what was initially developed to aid System Administration, this service went further to give a consistent enterprise view of the SCT infrastructure. By coupling multiple data sources, and incorporating resilience for points of failure, this service will become an essential tool for responding to power failures and resuming services in a quick and efficient manner. Until now, System Administrators would need to take a serial console to each individual PDU to power ports back up one by one; but this process was slow and required meticulous planning to ensure services were resumed in the correct sequence. Building a service like this also encourages best practices in order that it can provide the accurate information essential for System Administrators; as employing consistent naming conventions is very important, not just for the broader management perspective. Leveraging a modular design approach and jQuery UI interface, the service has proven easily maintained with minimal configuration overhead.

The SCT PDU & IPMI Management Portal has already proved useful for SCT System Administrators and new ideas on features it can provide are already being discussed, to further enhance its functionality. It is actively being used to streamline hardware maintenance and the Operating System commissioning process. The following future developments have been defined:

- To ensure that if an IPMI device is unavailable it does not slow down the loading of a page
- Dynamic display and updating of tab pane power status
- Emergency power-off capability for all PDU ports not connected to network infrastructure
- Provide a self contained remote client

9. Appendices

Appendix A: query()

All database interactivity is made by calling the query function. This takes 4 mandatory parameters and one optional one if you want to update an entry.

```
function query($db,$columns,$match,$criteria,$update = "SELECT") {
  $configfile = '/etc/cableguy.conf' ;
  $reply = array();
  $iniarray = parse_ini_file($configfile)
    or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;
  $dbloc = $iniarray["$db"];
  $lines = file($dbloc);
  if($update != "SELECT") {
    $headers = array_shift($lines);
    $header = array_map('trim', explode("|||",$headers));
    $match = array_search($match,$header);
    $dbtmp = $dbloc . ".tmp";
    $fh = fopen($dbtmp, 'w');
    fwrite($fh, $headers);
    foreach ($lines as $line) {
      $int = array_map('trim',explode("|||",$line));
      if ($criteria == $int[$match]) {
        $string = "";
        foreach ($header as $h) {
          if ($h == $columns) {
            $string .= $update . "|||";
          } else {
            $hnum = array_search($h,$header);
            $string .= $int[$hnum] . "|||";
          }
        }
        $string = rtrim($string,"|");
        $string .= "\n";
        fwrite($fh,$string);
      } else {
        fwrite($fh,$line);
      }
    }
    fclose($fh);
    exec("mv -f $dbtmp $dbloc");
    return;
  }
…
…
```

- $db = database table we wish to query ‘FROM’
- $columns = one or many column values we wish to ‘SELECT’ in our search
- $match = the column we are doing our ‘WHERE’ on
- $criteria = the search criteria ‘=’ or ‘LIKE’
- $update = the new value we wish to ‘UPDATE’ our $column with

The code above shows what happens when we pass in a fifth parameter and want to update a field. The first thing it does is read the ‘table’ file we’re requesting to search through into memory, then shift off the first array row into a ‘headers’ array so we have an index of columns. Searching through this array with our $match parameter will give us an index to work with when iterating through the rest of the table. As we’re updating an entry, we read out of the live cache file and write into a temp file for every line which does not match the criteria. If we do match the $criteria, then a new string is formed from the fields which we are not modifying and an updated value of the field we are modifying. That then is inserted into the table instead. After we have finished with the file, it is copied over the original cache file and the function ends.

If we are searching through the table and require multiple columns then the code runs as follows:

```
$header = array_map('trim', explode("|||",array_shift($lines)));
$match = array_search($match,$header);
if (is_array($columns) || is_array($criteria)) {
  if (is_array($columns)) {
    foreach ($columns as $column){
```

```
      $cols[] = array_search($column,$header);
    }
  }
  if (strpbrk($criteria,"%")) {
    $criteria = preg_replace("/[\\%]/","",$criteria);
    trim($criteria);
    $i=0;
    foreach($lines as $line) {
      $int = array_map('trim',explode("|||",$line));
      if (strstr($int[$match],$criteria)) {
        foreach ($cols as $c) {
          $string[] = $int[$c];
        }
        $reply[$i] = $string;
        unset($string);
        $i++;
      }
    }
  } else {
    $i=0;
    foreach($lines as $line) {
      $int = array_map('trim',explode("|||",$line));
      if ($criteria == $int[$match]) {
        foreach ($cols as $c) {
          $string[] = $int[$c];
        }
        $reply[$i] = $string;
        unset($string);
        $i++;
      }
    }
  }
}
```

An array of 'columns' is generated for each column we have requested and as we are returning an array of column values in our reply, the code is different enough that it has its own if statement compared to requesting one column value.

```
else {
  $columns = array_search($columns,$header);
  if (strpbrk($criteria,"%")) {
    $criteria = preg_replace("/[\\%]/","",$criteria);
    trim($criteria);
    foreach($lines as $line) {
      $int = array_map('trim',explode("|||",$line));
      if (strstr($int[$match],$criteria)) {
        $reply[] = $int[$columns];
      }
    }
  } else {
    foreach($lines as $line) {
      $int = array_map('trim',explode("|||",$line));
      if ($criteria == $int[$match]) {
        $reply[] = $int[$columns];
      }
    }
  }
}
if (isset($reply)){
  return $reply;
} else {
  return null;
}
```

If our $criteria has a % sign in it we do a different type of string comparison to ensure that partial matches can be returned from the function. Finally, in case we do not find any results, a null value can be returned.

Appendix B: readPDU()

```
function readPDU($pdu) {
  $sql = "SELECT ip FROM host where expected_hostname = '$pdu'";
  $result = query("mysqlhost","ip","expected_hostname",$pdu);
  $ip = $result[0];
  $ports = snmprealwalk("$ip","public", ".1.3.6.1.4.1.318.1.1.12.3.1.4");
  $ports = array_values($ports);
  $count = trim(($ports[0]), "INTEGER: ");
  $results = array_values(snmprealwalk("$ip","public", ".1.3.6.1.4.1.318.1.1.12.3.3.1.1.4"));
  for($i=0;$i<$count;$i++){
    $j=$i+1;
    if (isset($results[$i])) {
      $result = trim(($results[$i]), "INTEGER: ");
      if ($result == 2) {
        $status = "off";
      } elseif ($result == 1) {
        $status = "on";
      } else {
        $status = "unsupported";
      }
    } else {
      $status = "unsupported";
    }
    $values[$j] = $status;
  }
  return($values);
}
```

This function takes a PDU name as its parameter and returns an array of each port status as its result.

```
Array
(
    [1] => on
    [2] => on
    [3] => off
    [4] => on
    [5] => on
    [6] => on
    [7] => on
    [8] => off
    [9] => on
    [10] => on
    [11] => off
    [12] => off
    [13] => off
    [14] => off
    [15] => on
    [16] => off
    [17] => off
    [18] => on
    [19] => on
    [20] => off
    [21] => off
    [22] => off
    [23] => on
    [24] => off
)
```

In order to determine which IP address, as PDUs do not have fully qualified domain names and as a result are not stored in DNS, it must query the hostname database which has a record of private IP addresses associated with PDU names. Once this has been obtained, an snmpwalk query is performed using a vendor specific Object Identifier (OID) which will determine the number of ports contained on a PDU and the power status of each. The returned value from the PDU needs to be processed in order to provide a value that can be processed by the rest of the service without further conversion.

Appendix C: getAllHosts() getAttachedComponents()

function getAllHosts()

```
function getAllHosts($pdu) {
  $pduID = getComponentIDs($pdu);
  $result = query("oracleport",array("port_number","connector_id","port_type"),"component_id",$pduID[0]);
  foreach ($result as $row) {
    if ($row[2] == 9) {
      if ($row[1] == 0) {
        $val[] = "EMPTY";
        $hosts[$row[0]] = $val;
        unset($val);
      } else {
        $hosts[$row[0]] = getAttachedComponents($row[1],$pduID[0]);
        unset($val);
      }
    }
  }
  return($hosts);
}
```

To obtain all the equipment connected to a named PDU, a search is done by matching the parameter's 'component_id' against the 'port' table, which describes all the devices connected to this component. Empty ports are filled with 'EMPTY' to ensure easier processing to display the results. The port_type is recorded to ensure that the devices are connected via a power cable and not any other type of connector. As this port table only contains component and connector 'ids', it is then necessary to call the 'getAttachedComponents' function to determine the names of connected devices based on the connector_id that joins them.

function getAttachedComponents()

```
function getAttachedComponents($connector_id,$parent_id) {
  $val = array();
  $result = query("oracleport",array("component_id","port_type"),"connector_id",$connector_id);
  foreach ($result as $row) {
    if (($row[0] != $parent_id) && ($row[1] == 3)) {
      $result2 = query("oraclecomp","component_name","component_id",$row[0]);
      foreach ($result2 as $row2) {
        $val[] = $row2;
      }
    }
  }
  if (!isset($val[0])) {
    $val[] = "EMPTY";
  }
  return($val);
}
```

This function takes the parent device (PDU) and the connector_id connected to a specific port and asks what is on the other end. If there is more than one component connected, all values are returned. The datatype of the returned value is an array.

Appendix D: getIPMIStatus()

```
function getIPMIStatus($hostname) {
  $result = query("mysqlhost","ip","expected_hostname","IPMI-$hostname");
  if ((isset($result)) && (count($result) > 0)) {
    $ip = $result[0];
    $configfile = '/etc/cableguy.conf' ;
    $iniarray = parse_ini_file($configfile)
      or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;
    $user = $iniarray['ipmiuser'];
    $pass = $iniarray['ipmipass'];
    $ipmistat = exec("/usr/bin/ipmitool -H $ip -U $user -P $pass power status", $alloutput, $exit_code);
    if($exit_code == 0) {
      $ipmi = trim($ipmistat);
      if (preg_match('/Chassis Power is (\S+).*/', $ipmi, $match) ) {
        return "$match[1]";
      } else {
        return "BAAD";
      }
    } else {
      return "timeout";
    }
  } else {
    return "unsupported";
  }
}
```

To determine the IPMI power status of a hostname, a check is first done within the hostname database to see whether it has a valid IPMI device configured. This is indicated by a hostname entry in the database of IPMI-[hostname]. Without this, the function will return the string ‘unsupported’. If there is a configured device, it is then queried to return its power status. This is parsed to return ‘on’ or ‘off’, or if the device times out ‘timeout’.

Appendix E: setPDU() setIPMIStatus() setVM()

function setPDU()

```
function setPDU($pdu, $port, $action) {
  $dnlist = '/etc/dn-list';
  $lines = file($dnlist);
  $authorized = FALSE;
  $admin = (isset($_SERVER["SSL_CLIENT_S_DN"])) ? $_SERVER["SSL_CLIENT_S_DN"] : "Unknown";
  for($i=0;$i<count($lines);$i++) {
    if ($admin == trim($lines[$i])) {
      $authorized = TRUE;
    }
  }
  if (isset($_SERVER['REMOTE_ADDR'])) {
    $remotehost = trim(gethostbyaddr($_SERVER['REMOTE_ADDR']));
  } else {
    $remotehost = "EMPTY";
  }
  if ($remotehost == "ks2.esc.rl.ac.uk") {
    $authorized = TRUE;
  }
  if ($authorized) {
    $actions = array( "on" => '1', "off" => '2' );
    $result = query("mysqlhost","ip","expected_hostname",$pdu);
    $ip = $result[0];
    $oid = ".1.3.6.1.4.1.318.1.1.12.3.3.1.1.4.";
    $results = snmpset("$ip","RamB0ess","$oid$port","i","$actions[$action]");
    return $results;
  }
}
```

To change the value of a PDU port, the setPDU function requires three parameters; the PDU name, the port in question, and the status it should be changed to. An additional DN check is made for security reasons before allowing the function to continue. Upon success, the hostname database is queried to determine the PDU IP and then an OID string is formed using the provided power status it should be changed to. The result of this operation is returned.

function setIPMIStatus()

```
function setIPMIStatus($hostname,$action) {
  $result = query("mysqlhost","ip","expected_hostname","IPMI-$hostname");
  if ((isset($result)) && (count($result) > 0)) {
    $ip = $result[0];
    $configfile = '/etc/cableguy.conf' ;
    $iniarray = parse_ini_file($configfile)
      or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;
    $user = $iniarray['ipmiuser'];
    $pass = $iniarray['ipmipass'];
    $ipmistat = exec("/usr/bin/ipmitool -H $ip -U $user -P $pass power $action", $alloutput, $exit_code);
    if($exit_code == 0) {
      $ipmi = trim($ipmistat);
      return $ipmi;
    } else {
      return "$exit_code";
    }
  } else {
    return "unsupported";
  }
}
```

Changing the value of an IPMI device requires the ipmitool function and simply requires the hostname in question and action required as parameters. If the action is successful, the output of the command is returned, if there is a problem, an array of the error message is returned, if there is no IPMI device configured, the string ‘unsupported’ is returned.

function setVM()

```
function setVM($host, $guest, $state){
  $configfile = '/etc/cableguy.conf' ;
  $iniarray = parse_ini_file($configfile)
      or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;
  $vmuser = $iniarray["vmuser"];
  $vmpass = $iniarray["vmpass"];
  $guest .= "%";
  $vmx = query("vmrack",array("path","host"),"guest",$guest);
  if ($vmx[0][1] == $host) {
    $path = $vmx[0][0];
    $path = preg_replace("/[\\(]/","\\\\(",$path);
    $path = preg_replace("/[\\)]/","\\\\)",$path);
    $result = exec("vmware-cmd --server $host --username $vmuser --password $vmpass $path
$state",$alloutput, $exit_code);
    return($alloutput);
  } else {
    $string[] = "fuzzy match\n";
    return($string);
  }
}
```

For Virtual Machines, we need to know the host it is running on, the VM name and the power state we wish to update it to. Before a command can be issued, it is necessary to find the 'path' where the VM is stored on the host. This is done by querying the VM database to find its path and then using this as a parameter in the vmware-cmd command. The output of the command is returned.

Appendix F: findHostRack()

```
function findHostRack($hostname){
  # Finds the first PDU that a host is connected to then determines the rack
  $hostid = getComponentIDs($hostname);
  foreach ($hostid as $component_id) {
    $result = query("oracleport",array("connector_id","port_type"),"component_id",$component_id);
    $component_name = query("oraclecomp","component_name","component_id",$component_id);
    $component = $component_name[0];
    foreach ($result as $res){
      if (($res[1] == '3')) {
        $value = $res[0];
      }
    }
    if (isset($value) && ($value != 0)) {
      $result2 = query("oracleport",array("component_id","port_type"),"connector_id",$value);
      foreach ($result2 as $res2) {
        if ($res2[1] == '9') {
          $value = $res2[0];
        }
      }
      $result3 = query("oraclecomp","rack_id","component_id",$value);
      $rack = query("oraclerack","rack_name","rack_id",$result3[0]);
      $values[] = array("$component","$rack[0]");

    } else {

    }
  }
  return($values);
}
```

As part of the search engine, a query needs to be made to determine the rack which a passed hostname is located within. Firstly the component_id of the $hostname parameter is determined, which may be return more than one value if a wildcard search is being performed. For each of the returned component_ids, a search is made to find the connector_id that goes to a power supply, and then if it is successful, the PDU which is on the other end. Once a PDU has been determined, the Rack which it is located in is then searched for and returned. An array of all results is returned.

Appendix G: readVMs() readVMStatus() readLiveStatus()

function readVMs()

```
function readVMs($host) {
  $result = query("vmrack",array("guest","path"),"host",$host);
  return($result);
}
```

This very simple function parses the VM database table to find all the connected hosts and their respective paths for a given ESX hostname.

function readVMStatus()

```
function readVMstatus($vmx, $host) {
  $result = query("vmrack","state","path",$vmx);
  return($result[0]);
}
```

Similarly, as we cache the VM power status in a database, this can be determined very easily via the above function. To ensure no erroneous information is produced, we search by VM path and the ESX host it is believed to be connected to.

function readLiveStatus()

```
function readLiveStatus($vmx, $host) {
  $configfile = '/etc/cableguy.conf' ;
  $iniarray = parse_ini_file($configfile)
      or trigger_error("Unable to read $configfile, does it have the correct permissions?") ;
  $vmuser = $iniarray["vmuser"];
  $vmpass = $iniarray["vmpass"];
  $escvmx = $vmx;
  $escvmx = preg_replace("/[\\(]/","\\\\(",$escvmx);
  $escvmx = preg_replace("/[\\)]/","\\\\)",$escvmx);
  $result = exec("vmware-cmd --server $host --username $vmuser --password $vmpass $escvmx
getstate",$alloutput,$exit_code);
  $state = array_map('trim', explode(" = ",$result));
  query("vmrack","state","path",$vmx,$state[1]);
  return($state[1]);
}
```

To determine realtime VM power status, which is used for the caching function, we need to know the path of the VM and the host it is connected to. This function ensures that any control characters are escaped, such as brackets, and then calls the vmware-cmd application. The result of this function is then parsed and a query function is executed with five parameters, which will update the record cached in the database.

10. References

[1] STFC e-Science Centre: Scientific Computing Technology Group
http://sct.esc.rl.ac.uk

[2] Red Hat Enterprise Linux Kickstart
http://fedoraproject.org/wiki/Anaconda/Kickstart

[3] Simple Network Management Protocol
http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

[4] jQuery website and documentation
http://jquery.com/

[5] jQuery UI website and documentation
http://jqueryui.com/

[6] Intelligent Platform Management Interface
http://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface

[7] VMware ESX Virtualisation
http://www.vmware.com/products/esx/

[8] X.509 Certificates
http://en.wikipedia.org/wiki/X.509

[9] UK e-Science Certificate Authority
https://ca.grid-support.ac.uk/cgi-bin/pub/pki?cmd=getStaticPage&name=index

[10] vSphere CLI
http://www.vmware.com/pdf/vsphere4/r40/vsp_40_vcli.pdf