University of Nevada

Reno

Beyond Monitoring: Proactive Server Preservation in an HPC Environment

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

by

Chad E. Feller

Dr. Frederick C. Harris, Jr., Thesis advisor

May, 2012

University of Nevada, Reno
Statewide • Worldwide

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

CHAD E. FELLER

entitled

**Beyond Monitoring:
Proactive Server Preservation In An
HPC Environment**

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr., Ph. D., Advisor

Sergiu M. Dascalu, Ph. D., Committee Member

Karen A. Schlauch, Ph. D., Graduate School Representative

Marsha H. Read, Ph. D., Dean, Graduate School

May, 2012

Abstract

Monitoring has long been the challenge of a server administrator. Monitoring disk health, system load, network congestion, and environmental conditions like temperature are all things that can be tied into monitoring systems. Monitoring systems vary in scope and capabilities, and many can fire off alerts for just about any configuration. The sysadmin then has the responsibility of weighing the alert and deciding if and when to act. In a High Performance Computing (HPC) environment, some of these failures can have a ripple effect, affecting a larger area than the physical problem. Furthermore, some temperature and load swings can be more drastic in an HPC environment than they would be otherwise. Because of this a timely, measured response is critical. When a timely response is not possible, conditions can escalate rapidly in an HPC environment, leading to component failure. In this situation, an intelligent, automatic, measured response is critical. Here we present such a system, a novel approach to server monitoring using integrated server hardware operating independently of the operating sytem, and capable not only of monitoring temperatures, but also automatically responding to temperature events. Our proactive response system leverages standard HPC software and integrated server hardware. It is designed to intelligently respond to temperature events from a High Performance Computing perspective, looking at both compute jobs and server hardware.

Acknowledgements

I would like to thank my wife, Veronica, for being an amazing support and encouragement through all of this, my kids, for understanding when I couldn't always spend time with them, and my parents and grandparents for relentlessly prodding me over the years to finally finish this.

I would also like to thank my advisor, Dr. Frederick Harris, for guidance, the more than occasional "nudge", to keep moving throughout all of this, and for bearing with me through my numerous thesis ideas over the years; to Dr. Karen Schlauch, Director of the Center for Bioinformatics, for being supportive in my graduate endeavors, and giving me opportunities to do good science while completing my thesis; and to Dr. Sergiu Dascalu for being on my commitee.

I would like to thank my good friend, Derek Eiler, for taking time out of his busy schedule to provide valuable proofreading and feedback.

I would also like to thank Yukihiro Matsumoto ("Matz") for the Ruby programming language, David Heinemeier Hansson ("DHH") for Rails, and Tobias "Tobi" Oetiker for RRDtool. Without these fabulous languages and libraries to leverage, this project would have taken much longer.

This work was partially supported by NIH Grant Number P20 RR-016464 from the INBRE Program of the National Center for Research Resources.

Contents

List of Figures

Chapter 1

Introduction

A common, if not typical, approach to monitoring systems is to have a central host that monitors servers over the network. A most basic implementation would have a central monitoring host that pings the monitored server via ICMP to see if it responds, or sends requests to specified TCP/UDP ports to ensure that services are still running on those ports. This basic approach can answer the questions of whether a server is up or down, and whether or not specified services are available over the network. A more intelligent approach involves installing a daemon, or system service, on the monitored server, allowing the monitoring system to not only observe external behavior as in the basic model, but also to see what is going on inside of the monitored server. The benefit here is clear: CPU load and memory issues can be observed among other things such as process characteristics. The downside is that these daemons are typically OS specific, and themselves can fail for various reasons. They may be slow to respond if the monitored server is under heavy load, or can be terminated by the OS under memory pressure. Daemons are ineffective if the OS has a kernel panic due to a hardware or software failure.

Witnessing a server room temperature event quickly spiral out of control is what inspired the design behind this system. We had a server room where one of three A/C units failed to switch over to generator power after a power failure. Since it takes several minutes for the generators to spin up to full capacity, UPS units keep the servers and networking equipment running until the generators can take over. Due to cost, the UPS units do not also run the A/C system. The generators take roughly five to seven minutes to reach full operating capacity, during which time the temperature begins to rise. Once on generator power, and the A/C units switch back on, it takes several minutes before they begin to blow cold. Because of this, the management team (managers and system administrators) expect to see the server room temperature rise for approximately 10 minutes following a power outage.

In this particular situation, there had been several brownouts in rapid succession immediately before the power went out. It is believed that one of these brownouts damaged the unit that switches an A/C unit over to generator power, ultimately causing that one A/C unit to stay offline. Roughly 15 minutes after the loss of power, with two A/C units running, the management team could see that the rise in temperature had begun to slow, and expected to see the temperature begin to fall during a subsequent polling cycle, but this didn't happen. As is often the case, these types of events happen after hours, and this event was no different. Managers and system administrators are reluctant to go back into work unless they are absolutely certain it is necessary. After watching the temperature rise longer than it should, hoping that it will come down during the next polling cycle, the management team finally reacted.

By the time members of the management team got back to the server room that night, they found a room that was well over 100 degrees. They knew there wasn't time to log into each console and cleanly shut down each server, nor was there enough time to hold each power button for five seconds. The team was in disaster control mode, and began to pull the power cord from the back of each server. A hard shutdown like this isn't ideal, but it is better than the risk of completely loosing hardware.

In spite of this, about a dozen HPC servers were lost, not to mention the compute jobs which were running on them.

The compute density and associated heat production from HPC equipment caused the temperature to rise much more quickly than it would have had only traditional computing equipment been in this server room, giving the management team little time to effectively evaluate the situation and react. Because time is critical when dealing with HPC systems, we developed a proactive system to automatically and intelligently deal with events just like this.

The remainder of this thesis is structured as follows: Chapter 2 presents a few select examples of other monitoring systems. Chapter 3 presents a novel design from a high level overview, along with an analysis of the components leveraged by our design. Chapter 4 takes an in depth look at the implementation, first from a high level, and then on a component by component basis. The section finishes with a walkthrough of the algorithm. Chapter 5 concludes with the current state of the project, and Chapter 6 discusses future work.

Chapter 2

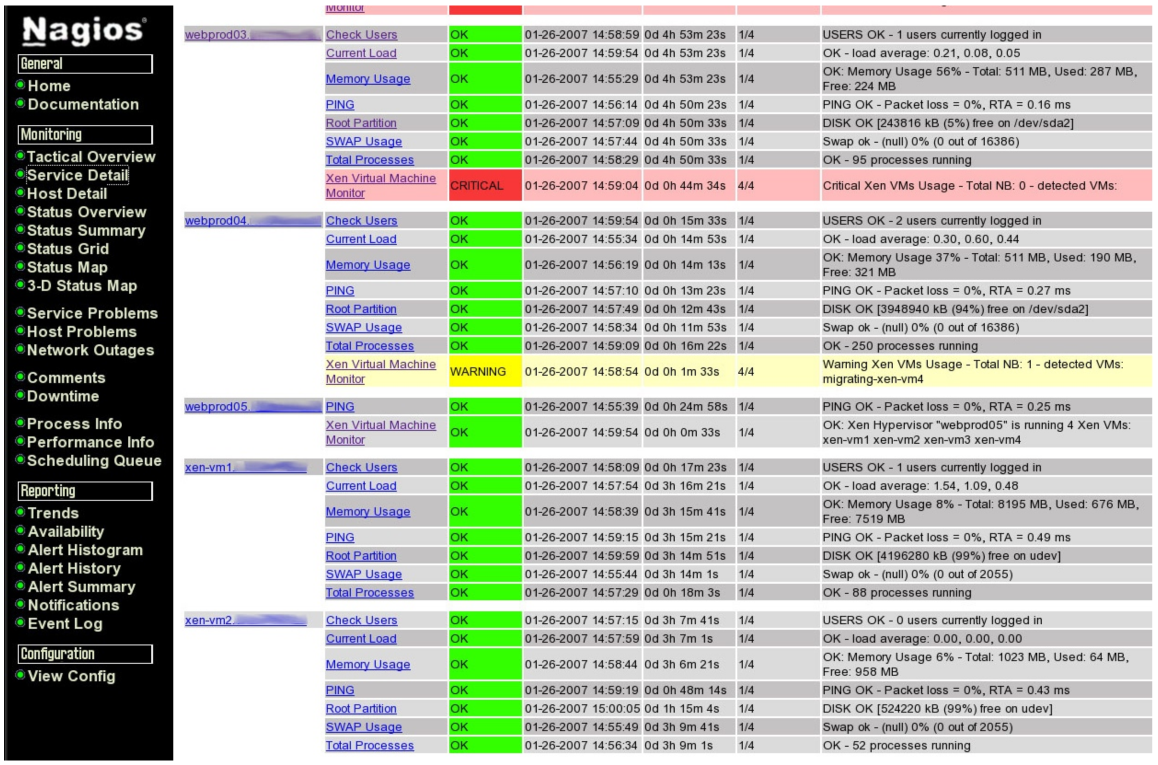
Background and Related Work

2.1 System and Network Monitors

There are several open source system monitoring and network monitoring software solutions freely available over the internet. There are yet more available if commercial offerings are also considered. However, many existing monitoring solutions are either overly broad or narrowly focused. Here we will take a look at some of the existing solutions.

Nagios

According to the Nagios website, “Nagios is a powerful monitoring system that enables organizations to identify and resolve IT infrastructure problems before they affect critical business processes.” [29, 63] In practice, Nagios is a monitoring tool capable of monitoring a range of network and system services, as well as host resources via plugins and daemons that reside on the remote host. It also allows the system administrator to provide a custom response to an event. The system administrator writes custom event handlers which may be as simple as “text this group of people” or as elaborate as "write this message to the logs, and reboot the server". Nagios uses a flat file for its database backend by default. Nagios can present its data several different ways, and a default view can be seen in Figure 2.1.

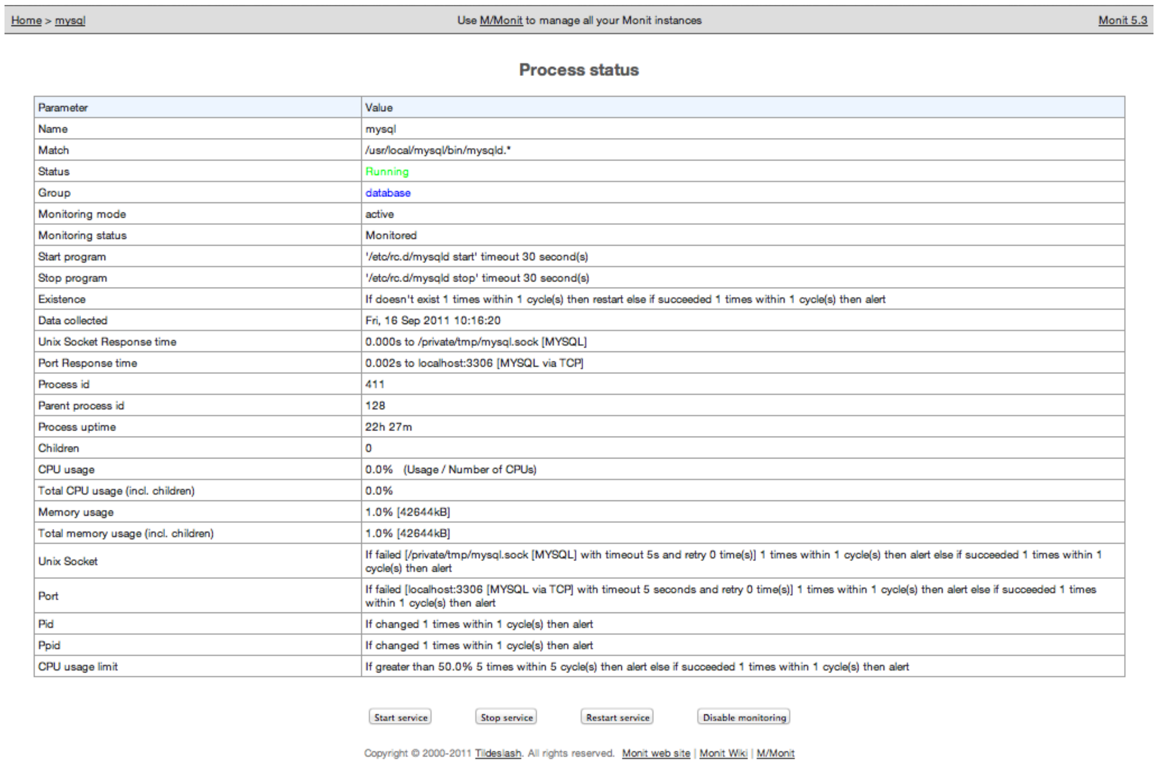


	Monitor					
webprod03.[redacted]	Check Users	OK	01-26-2007 14:58:59	0d 4h 53m 23s	1/4	USERS OK - 1 users currently logged in
	Current Load	OK	01-26-2007 14:59:54	0d 4h 53m 23s	1/4	OK - load average: 0.21, 0.08, 0.05
	Memory Usage	OK	01-26-2007 14:55:29	0d 4h 53m 23s	1/4	OK: Memory Usage 56% - Total: 511 MB, Used: 287 MB, Free: 224 MB
	PING	OK	01-26-2007 14:56:14	0d 4h 50m 23s	1/4	PING OK - Packet loss = 0%, RTA = 0.16 ms
	Root Partition	OK	01-26-2007 14:57:09	0d 4h 50m 33s	1/4	DISK OK [243816 kB (5%) free on /dev/sda2]
	SWAP Usage	OK	01-26-2007 14:57:44	0d 4h 50m 33s	1/4	Swap ok - (null) 0% (0 out of 16386)
	Total Processes	OK	01-26-2007 14:58:29	0d 4h 50m 33s	1/4	OK - 95 processes running
	Xen Virtual Machine Monitor	CRITICAL	01-26-2007 14:59:04	0d 0h 44m 34s	4/4	Critical Xen VMs Usage - Total NB: 0 - detected VMs:
webprod04.[redacted]	Check Users	OK	01-26-2007 14:59:54	0d 0h 15m 33s	1/4	USERS OK - 2 users currently logged in
	Current Load	OK	01-26-2007 14:55:34	0d 0h 14m 53s	1/4	OK - load average: 0.30, 0.60, 0.44
	Memory Usage	OK	01-26-2007 14:56:19	0d 0h 14m 13s	1/4	OK: Memory Usage 37% - Total: 511 MB, Used: 190 MB, Free: 321 MB
	PING	OK	01-26-2007 14:57:10	0d 0h 13m 23s	1/4	PING OK - Packet loss = 0%, RTA = 0.27 ms
	Root Partition	OK	01-26-2007 14:57:49	0d 0h 12m 43s	1/4	DISK OK [3948940 kB (94%) free on /dev/sda2]
	SWAP Usage	OK	01-26-2007 14:58:34	0d 0h 11m 53s	1/4	Swap ok - (null) 0% (0 out of 16386)
	Total Processes	OK	01-26-2007 14:59:09	0d 0h 16m 22s	1/4	OK - 250 processes running
	Xen Virtual Machine Monitor	WARNING	01-26-2007 14:58:54	0d 0h 1m 33s	4/4	Warning Xen VMs Usage - Total NB: 1 - detected VMs: migrating-xen-vm4
webprod05.[redacted]	PING	OK	01-26-2007 14:55:39	0d 0h 24m 58s	1/4	PING OK - Packet loss = 0%, RTA = 0.25 ms
	Xen Virtual Machine Monitor	OK	01-26-2007 14:59:54	0d 0h 0m 33s	1/4	OK: Xen Hypervisor "webprod05" is running 4 Xen VMs: xen-vm1 xen-vm2 xen-vm3 xen-vm4
xen-vm1.[redacted]	Check Users	OK	01-26-2007 14:58:09	0d 0h 17m 23s	1/4	USERS OK - 1 users currently logged in
	Current Load	OK	01-26-2007 14:57:54	0d 3h 16m 21s	1/4	OK - load average: 1.54, 1.09, 0.48
	Memory Usage	OK	01-26-2007 14:58:39	0d 3h 15m 41s	1/4	OK: Memory Usage 8% - Total: 8195 MB, Used: 676 MB, Free: 7519 MB
	PING	OK	01-26-2007 14:59:15	0d 3h 15m 21s	1/4	PING OK - Packet loss = 0%, RTA = 0.49 ms
	Root Partition	OK	01-26-2007 14:59:59	0d 3h 14m 51s	1/4	DISK OK [4196280 kB (99%) free on udev]
	SWAP Usage	OK	01-26-2007 14:55:44	0d 3h 14m 1s	1/4	Swap ok - (null) 0% (0 out of 2055)
	Total Processes	OK	01-26-2007 14:57:29	0d 0h 18m 3s	1/4	OK - 88 processes running
xen-vm2.[redacted]	Check Users	OK	01-26-2007 14:57:15	0d 3h 7m 41s	1/4	USERS OK - 0 users currently logged in
	Current Load	OK	01-26-2007 14:57:59	0d 3h 7m 1s	1/4	OK - load average: 0.00, 0.00, 0.00
	Memory Usage	OK	01-26-2007 14:58:44	0d 3h 6m 21s	1/4	OK: Memory Usage 6% - Total: 1023 MB, Used: 64 MB, Free: 958 MB
	PING	OK	01-26-2007 14:59:19	0d 0h 48m 14s	1/4	PING OK - Packet loss = 0%, RTA = 0.43 ms
	Root Partition	OK	01-26-2007 15:00:05	0d 1h 15m 4s	1/4	DISK OK [524220 kB (99%) free on udev]
	SWAP Usage	OK	01-26-2007 14:55:49	0d 3h 9m 41s	1/4	Swap ok - (null) 0% (0 out of 2055)
	Total Processes	OK	01-26-2007 14:56:34	0d 3h 9m 1s	1/4	OK - 52 processes running

Figure 2.1: The "Service Detail" view in Nagios [19].

Monit

Monit is a monitoring tool, written in C, primarily designed to monitor system (daemon) processes and take predefined actions should certain conditions happen. In fact, Monit's site says this is what sets them apart: "In difference to many monitoring systems, Monit can act if an error situation should occur, e.g.; if sendmail is not running, Monit can start sendmail again automatically or if apache is using too many resources (e.g. if a DoS attack is in progress) Monit can stop or restart apache and send you an alert message." [6] Additionally, Monit can be used to check system resources, much like other tools. A screenshot of Monit monitoring a MySQL process can be seen in Figure 2.2.



Process status

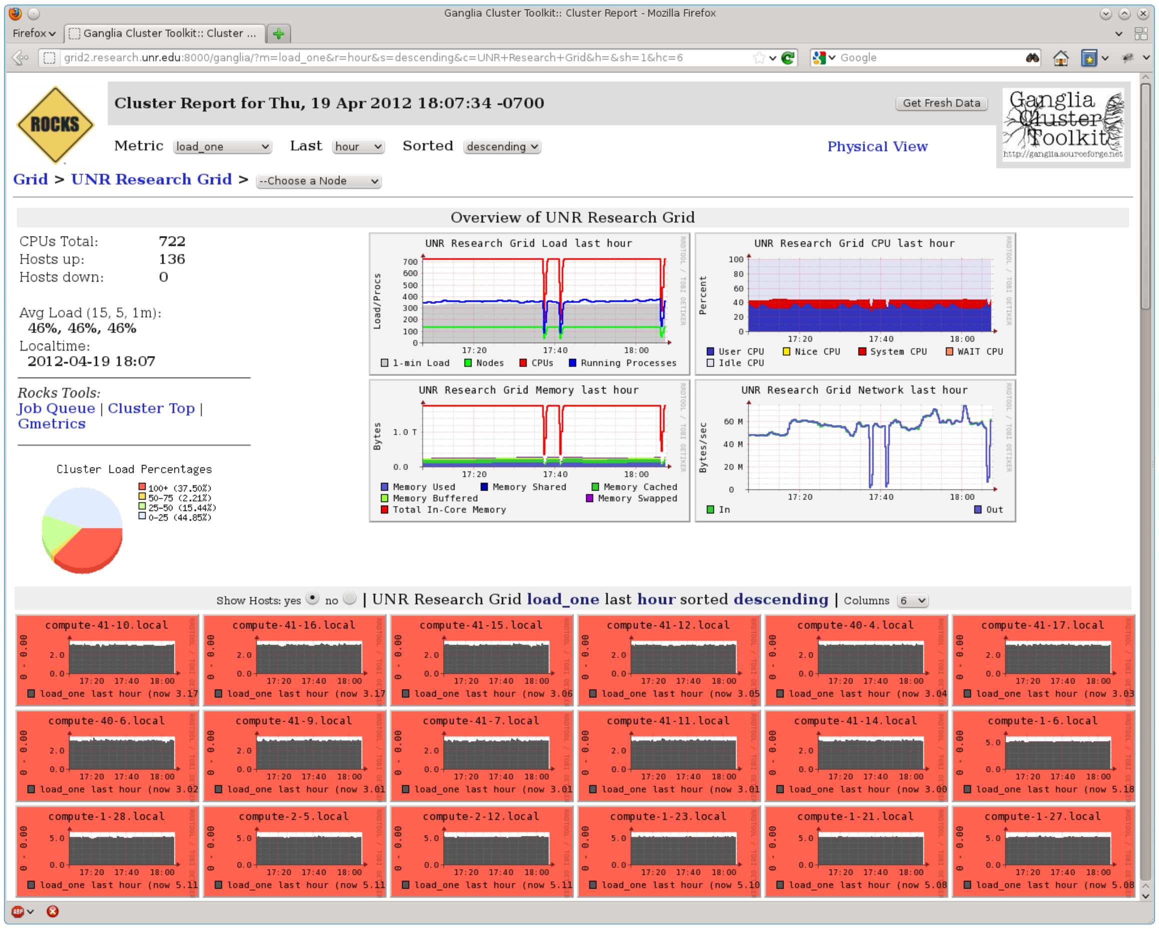
Parameter	Value
Name	mysql
Match	/usr/local/mysql/bin/mysqld.*
Status	Running
Group	database
Monitoring mode	active
Monitoring status	Monitored
Start program	'/etc/rc.d/mysqld start' timeout 30 second(s)
Stop program	'/etc/rc.d/mysqld stop' timeout 30 second(s)
Existence	If doesn't exist 1 times within 1 cycle(s) then restart else if succeeded 1 times within 1 cycle(s) then alert
Data collected	Fri, 16 Sep 2011 10:16:20
Unix Socket Response time	0.000s to /private/tmp/mysql.sock [MYSQL]
Port Response time	0.002s to localhost:3306 [MYSQL via TCP]
Process id	411
Parent process id	128
Process uptime	22h 27m
Children	0
CPU usage	0.0% (Usage / Number of CPUs)
Total CPU usage (incl. children)	0.0%
Memory usage	1.0% [42644kB]
Total memory usage (incl. children)	1.0% [42644kB]
Unix Socket	If failed [/private/tmp/mysql.sock [MYSQL] with timeout 5s and retry 0 time(s)] 1 times within 1 cycle(s) then alert else if succeeded 1 times within 1 cycle(s) then alert
Port	If failed [localhost:3306 [MYSQL via TCP] with timeout 5 seconds and retry 0 time(s)] 1 times within 1 cycle(s) then alert else if succeeded 1 times within 1 cycle(s) then alert
Pid	If changed 1 times within 1 cycle(s) then alert
Ppid	If changed 1 times within 1 cycle(s) then alert
CPU usage limit	If greater than 50.0% 5 times within 5 cycle(s) then alert else if succeeded 1 times within 1 cycle(s) then alert

Copyright © 2000-2011 Tildeslash. All rights reserved. Monit web site | Monit Wiki | M/Monit

Figure 2.2: A Process Status view in Monit [23].

Ganglia

Ganglia [9, 66] is a distributed system monitoring tool designed to log and graph CPU, memory, and network loads, both historically and in real time. It is used primarily in HPC cluster environments. Ganglia was written modularly in several languages including C, Perl, Python, and PHP. Unlike Nagios and Monit, Ganglia isn't designed to do anything beyond reporting collected data. That is, it won't alert the system administrator if certain events occur. A Ganglia view of our local HPC grid can be seen in Figure 2.3.


Cluster Report for Thu, 19 Apr 2012 18:07:34 -0700

Metric load_one Last hour Sorted descending

Grid > UNR Research Grid > --Choose a Node

Overview of UNR Research Grid

CPUs Total:	722
Hosts up:	136
Hosts down:	0

Avg Load (15, 5, 1m):
46%, 46%, 46%
Localtime:
2012-04-19 18:07

Rocks Tools:
Job Queue | Cluster Top | Gmetrics

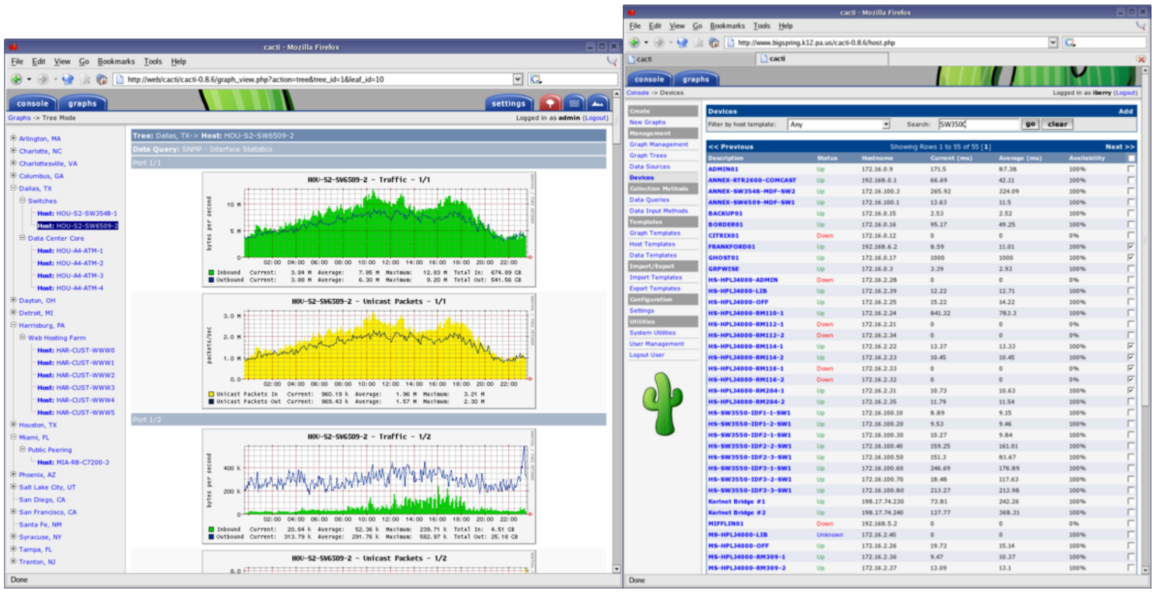
Show Hosts: yes [✓] no | UNR Research Grid **load_one** last **hour** sorted **descending** | Columns 6

compute-41-10.local (now 3.17	compute-41-16.local (now 3.17	compute-41-15.local (now 3.06	compute-41-12.local (now 3.05	compute-40-4.local (now 3.04	compute-41-17.local (now 3.03
compute-40-6.local (now 3.02	compute-41-9.local (now 3.01	compute-41-7.local (now 3.01	compute-41-11.local (now 3.01	compute-41-14.local (now 3.00	compute-1-6.local (now 5.18
compute-1-28.local (now 5.11	compute-2-5.local (now 5.11	compute-2-12.local (now 5.11	compute-1-23.local (now 5.10	compute-1-21.local (now 5.08	compute-1-27.local (now 5.08

Figure 2.3: A Ganglia view of our local HPC grid.

Cacti

Cacti [4] is a monitoring tool, written in PHP, designed to monitor, log and graph, both historically and real time network traffic. It is designed for, and commonly used to, monitor network switches and routers. However, Cacti can be configured to monitor just about any data source that can be stored in an RRD (Round Robin Database). Like Ganglia, it is only designed to report data that it collects, and doesn't support the ability to send alerts or custom responses. A view of Cacti monitoring graphs can be seen in Figure 2.4.

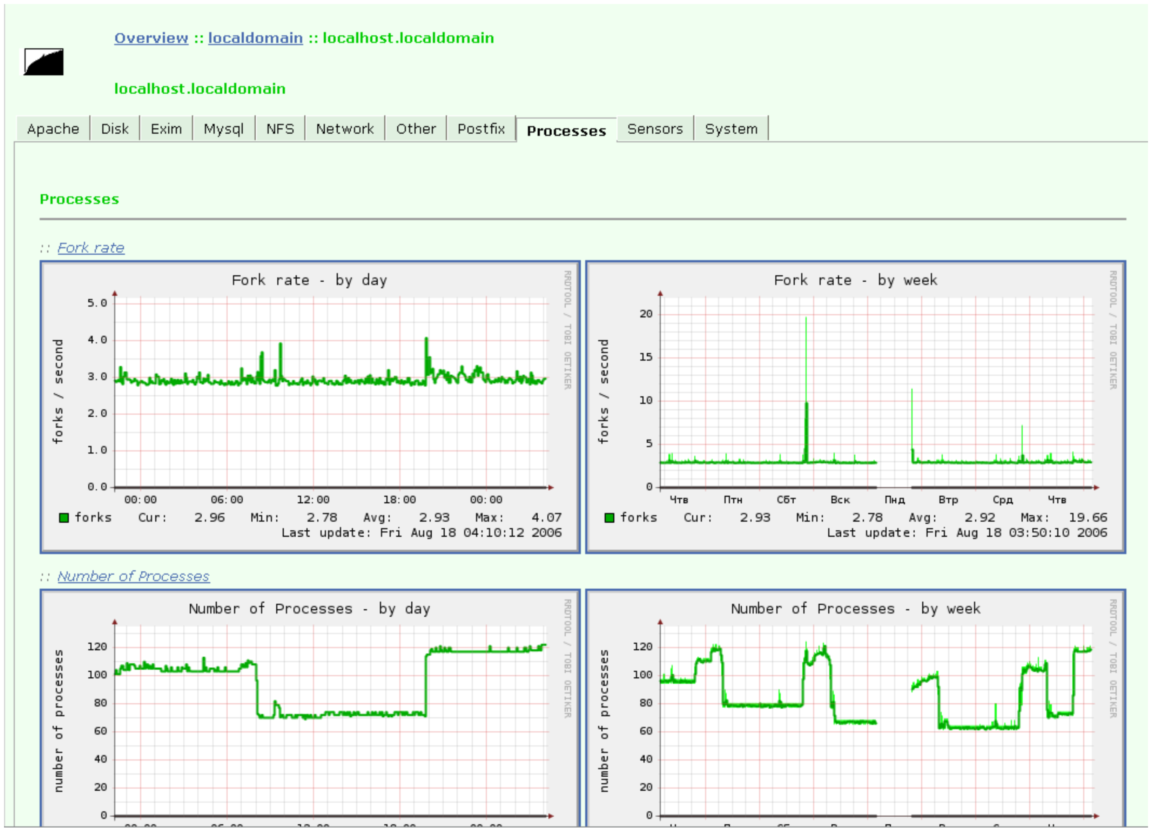

cacti - Mozilla Firefox
http://web/cacti/cacti-0.8.6/graph_view.php?action=tree&tree_id=1&leaf_id=10
console
graphs
settings
Tree: Dallas, TX-> Host: HOU-S2-SW6509-2
Data Query: SNMP - Interface Statistics
HOU-S2-SW6509-2 - Traffic - 1/1
HOU-S2-SW6509-2 - Unicast Packets - 1/1
HOU-S2-SW6509-2 - Traffic - 1/2
HOU-S2-SW6509-2 - Unicast Packets - 1/2
http://www.bigspring.k12.pa.us/cacti-0.8.6/host.php
Logged in as lberry (Logout)
Devices
Add
Filter by host template: Any
Search: SW350
go
clear
<< Previous
Showing Rows 1 to 55 of 55 [1]
Next >>

Description	Status	Hostname	Current (ms)	Average (ms)	Availability
ADMIN01	Up	172.16.0.9	171.5	87.38	100%
ANNEX-RTR2600-COMCAST	Up	192.168.0.1	66.69	42.11	100%
ANNEX-SW3548-MDF-SW2	Up	172.16.100.3	265.92	324.09	100%
ANNEX-SW6509-MDF-SW1	Up	172.16.100.1	13.63	11.5	100%
BACKUP01	Up	172.16.0.15	2.53	2.52	100%
BORDER01	Up	172.16.0.16	95.17	49.25	100%
CITRIX01	Down	172.16.0.12	0	0	0%
FRANKFORD01	Up	192.168.6.2	8.59	11.01	100%
GHOST01	Up	172.16.0.17	1000	1000	100%
GRPWISE	Up	172.16.0.3	3.39	2.93	100%
HS-HPLJ4000-ADMIN	Down	172.16.2.28	0	0	0%
HS-HPLJ4000-LIB	Up	172.16.2.39	12.22	12.71	100%
HS-HPLJ4000-OFF	Up	172.16.2.25	15.22	14.22	100%
HS-HPLJ4000-RM110-1	Up	172.16.2.24	841.32	783.3	100%
HS-HPLJ4000-RM112-1	Down	172.16.2.21	0	0	0%
HS-HPLJ4000-RM112-2	Down	172.16.2.34	0	0	0%
HS-HPLJ4000-RM114-1	Up	172.16.2.22	13.37	13.33	100%
HS-HPLJ4000-RM114-2	Up	172.16.2.23	10.45	10.45	100%
HS-HPLJ4000-RM116-1	Down	172.16.2.33	0	0	0%
HS-HPLJ4000-RM116-2	Down	172.16.2.32	0	0	0%
HS-HPLJ4000-RM204-1	Up	172.16.2.31	10.73	10.63	100%
HS-HPLJ4000-RM204-2	Up	172.16.2.35	11.79	11.54	100%
HS-SW3550-IDF1-1-SW1	Up	172.16.100.10	8.89	9.15	100%
HS-SW3550-IDF1-2-SW1	Up	172.16.100.20	9.53	9.46	100%
HS-SW3550-IDF2-1-SW1	Up	172.16.100.30	10.27	9.84	100%
HS-SW3550-IDF2-2-SW1	Up	172.16.100.40	159.25	161.01	100%
HS-SW3550-IDF2-3-SW1	Up	172.16.100.50	151.3	81.67	100%
HS-SW3550-IDF3-1-SW1	Up	172.16.100.60	246.69	176.89	100%
HS-SW3550-IDF3-2-SW1	Up	172.16.100.70	18.48	117.63	100%
HS-SW3550-IDF3-3-SW1	Up	172.16.100.80	213.27	213.98	100%
Karlnet Bridge #1	Up	198.17.74.230	73.81	242.26	100%
Karlnet Bridge #2	Up	198.17.74.240	137.77	368.31	100%
MIFFLIN01	Down	192.168.5.2	0	0	0%
MS-HPLJ4000-LIB	Unknown	172.16.2.40	0	0	100%
MS-HPLJ4000-OFF	Up	172.16.2.26	19.73	15.14	100%
MS-HPLJ4000-RM309-1	Up	172.16.2.36	9.47	10.37	100%
MS-HPLJ4000-RM309-2	Up	172.16.2.37	13.09	13.1	100%

Figure 2.4: Two common views of the Cacti monitoring interface [3].

Munin

Munin [25] is a monitoring tool, written in Perl, designed to monitor system and network performance. It aims to be as "plug and play" as possible, to "just work" out of the box. Additionally, Munin, like Cacti, presents both historical and current data, attempting to make it as easy as possible to see "what is different today" when performance issues arise. While Munin doesn't have the ability to send alerts like Nagios, Munin can integrate with Nagios to leverage its alert and response system [26]. A custom Munin screenshot can be seen in Figure 2.5.



```
■ forks   Cur:   2.96   Min:   2.78   Avg:   2.93   Max:   4.07
                          Last update: Fri Aug 18 04:10:12 2006
```

```
■ forks   Cur:   2.93   Min:   2.78   Avg:   2.92   Max:  19.66
                          Last update: Fri Aug 18 03:50:10 2006
```

Figure 2.5: A custom Munin view, representing the typical Munin dataset [27].

2.2 Environmental Monitors

Environmental monitors are commonly employed in server rooms to monitor humidity and temperature, two important elements of server room climate control. Sample environmental monitors can be seen in Figure 2.6. While temperature is a clear concern, humidity can also be an issue, as too much humidity in the air can cause condensation can occur. Too little humidity can enable the buildup of static electricity

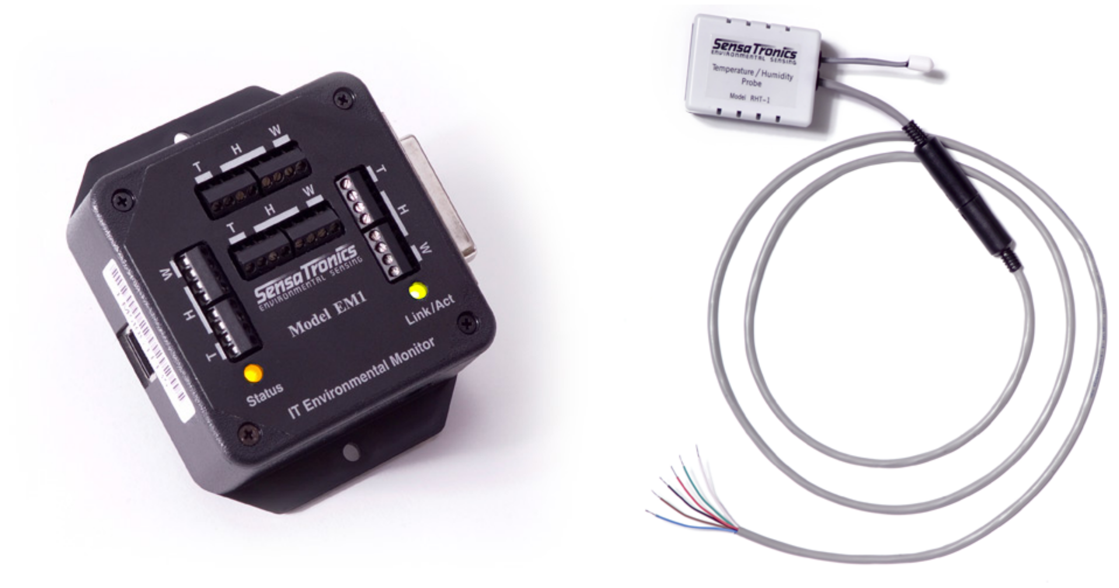


Figure 2.6: A network equipped environmental monitor [22], left, capable of supplying data to a larger monitoring system. A temperature and humidity sensor [49], right, is one of several sensors that can attach to this environmental monitor.

in the air.

Traditional environmental monitors are standalone units which may simply sound an audible alarm or even send out alert emails if they are network enabled. Environmental monitors are also available as network aware standalone units capable of feeding temperature points into other system monitoring software, whether they are proprietary or custom built solutions [22]. Simple temperature-only monitors are also available, many of which physically connect to a server, for instance via a USB dongle.

Chapter 3

A Proactive Approach

The motivation for this project comes from the fact that HPC environments are acutely susceptible to environmental failures, particularly cooling system failures. A cooling system failure in a server room with HPC equipment can spiral out of control much more quickly than other server rooms hosting more conventional computing equipment with intermittent loads such as simple mail and web servers.

There are a couple of reasons for this. First, HPC equipment is generally procured in very dense configurations (a high number of processor cores per rack unit [42]). Second, traditional computing equipment will generally only see momentary loads, not the type of sustained loads seen in Figure 3.1.

It is worth noting, however, that in this era of increased computing density due to virtualization, the difference in temperature output of HPC vs non-HPC equipment may begin to diminish.

If an environmental failure happens, particularly after working hours, and the system administrator's pager or cell phone goes off because because the environmental monitor sent out an alert, will the system administrator be able to get to a manage-

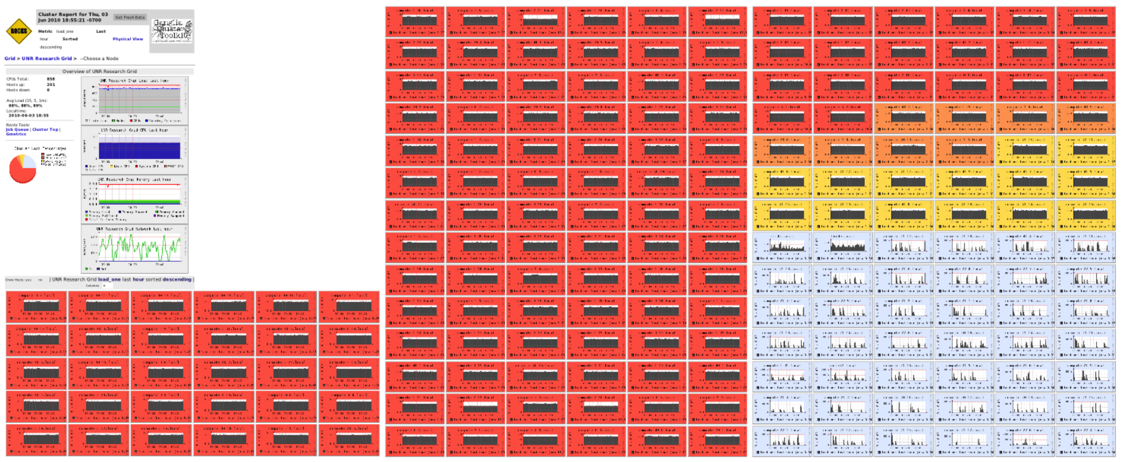


Figure 3.1: A Ganglia screenshot showing load in a typical HPC environment

ment console, login remotely, evaluate the situation, determine the best course of action, and then! execute that action before the temperature crests into the critical zone and causes hardware failure?

The fact that a commodity 1U [42] server can be ordered with more than 12 CPU cores makes the potential for heat generation greater than it was 4 years ago, when that same 1U server was only available with 4 CPU cores. The HVAC [12] should be upgraded to handle the additional capacity - if the capacity is not already there - but when the HVAC system fails, the temperature will rise much faster in that same server room than it would have a few years ago, giving the system administrator much less time to effectively respond.

Additionally, in an HPC environment compute jobs must be considered. Many HPC jobs run for days, weeks, or even months. Telling a researcher that they lost a month of compute time is neither easy, nor pleasant. In a server room with rapidly escalating temperatures, it is challenging for the system administrator to ascertain which jobs to try to save - if possible - and which to quickly shut down. The usual response is just “shut everything down now”.

But what if we could do this all more intelligently? This is the inspiration of our system that can automatically and intelligently respond to an environmental failure.

In thinking of how to implement this system, we took into account the fact that we needed to be able to monitor not only global cooling failures, but also localized cooling failures. Localized cooling failures may be caused by a bad fan, or perhaps a piece of debris - from newly unpackaged hardware, for instance - sucked into the front of a server rack obstructing airflow into one or more servers. To best detect localized failures, temperature sensors would need to be deployed on every server.

Computers have come with internal temperature sensors for some time now. Motherboard and CPU temperature sensors are quite common, and are even visible in the system BIOS on many basic desktop computers. Servers and high end workstations have several temperature sensors, including ambient temperature sensors that the operating system can read if necessary. However, allowing an outside resource to tap into this information typically requires a daemon process on the operating system to read the information. The system daemon of course, brings with it its own constraints, as mentioned in the introduction, so we wanted to leverage something else. Before discussing this idea further, an overview of some key components of the environment is necessary.

3.1 IPMI

An Intelligent Platform Management Interface [15, 1], commonly referred to as IPMI, or even as a BMC (Baseboard Management Controller), is a subsystem providing the system administrator with a method to communicate with and manage another device. It is specifically well suited for LOM, or “Lights Out Management” (also commonly called “Out-Of-Band Management”), tasks. An IPMI system is available for most mid to high end servers today, from all major vendors: Sun/Oracle, Dell, HP, and IBM, and on some high end, specialized workstations such as those provided by Fujitsu. These IPMI systems are known by such names as ILOM (Sun/Oracle), DRAC (Dell), iLO(HP), and RSA (IBM).

An IPMI system runs independently of the operating system and other hardware on the system, allowing the system administrator to interface with the server or workstation, even when it is powered off, or has locked up due to a kernel panic.

At a most basic level an IPMI system will allow a system administrator to query details like chassis power status, view event logs, query hardware configuration such as sensor information or FRU data, and turn the server or workstation on and off.

In recent IPMI implementations some vendors have built in a notification system capable of sending emails alerts to the system administrator if certain conditions occur, such as a sensor threshold being exceeded.

Many vendors add optional capabilities such as console redirection and remote drive access, allowing the system administrator to remotely access the device as though they were sitting in front of the system’s console. Vendors often make this functionality available through a web browser and/or standalone client software.

Sun provides basic functionality such as the ability to view logs, view sensor data and hardware information, view chassis power status, and turn the system on and off through a web browser. A Java Web Start program redirects video card output to the remote system administrator, and redirects keyboard and mouse activity from the remote system administrator to the server. It is also capable of redirecting a

CD-ROM or ISO image from the remote system administrator back to the server.

This allows the system administrator to sit down at their workstation and access a server hundreds of miles away as though sitting in front of it, turning it on, watching the BIOS POST, booting from a CD-ROM image stored on system administrator's local machine, an allowing the operating system to be reinstalled. This is all possible without the need for remote desktop or VNC software, and can be initiated even if the system is powered off.

The IPMI system may be accessed several different ways. We'll briefly discuss the most common ones here.

- It is available via serial console. This method is typical during initial IPMI system configuration, before the system is even turned on.
- It is also available via system software, locally on the machine, provided by a kernel driver. This method is handy for configuring the IPMI system after the operating system is installed and is running.
- It is also available over the network. This is typically achieved by sharing an interface with the system NIC, via what is known as side-band management, or a dedicated NIC giving it true "out-of-band" management.
- Whether available via side-band, or out-of-band methods, the IPMI system is typically communicated with over the network using tools that speak the IPMI messaging protocol. Many IPMI systems also run an SSH server by default, allowing communication over that protocol.

The IPMI product in use for the environment in this thesis is Sun's ILOM [13]. In this work, we'll be accessing it over the network. A picture of an ILOM can be seen in Figure 3.2.

Some vendors will refer to their IPMI system as a Baseboard Management Controller (BMC). Technically, the BMC is the core of the IPMI subsystem and is the microcontroller that ties the whole IPMI system together. See Figure 3.3.

3.2 Schedulers

Job schedulers form a key component of HPC environments and ensure fair and equal job scheduling: that no single job starves, and that no single job consumes all system resources. Two prominent schedulers are PBS (Portable Batch System) and SGE (Sun Grid Engine).

PBS is a loose term actually referring to a few different versions. The two main versions available today, and being actively developed are:

- TORQUE (Terascale Open-Source Resource and QUEue) Resource Manager [54], is a fork of what was OpenPBS [34], which is no longer actively developed [39].
- PBS Professional (PBS Pro) [40], a commercial offering.

SGE, although still widely used in current clusters, recently forked into multiple projects after the acquisition of Sun Microsystems by Oracle in 2010 [35]. Some of the prominent forks are:

- Oracle Grid Engine [37], a closed source commercial product from Oracle.
- Open Grid Scheduler [32], an open source offering based on the last open source release of SGE.

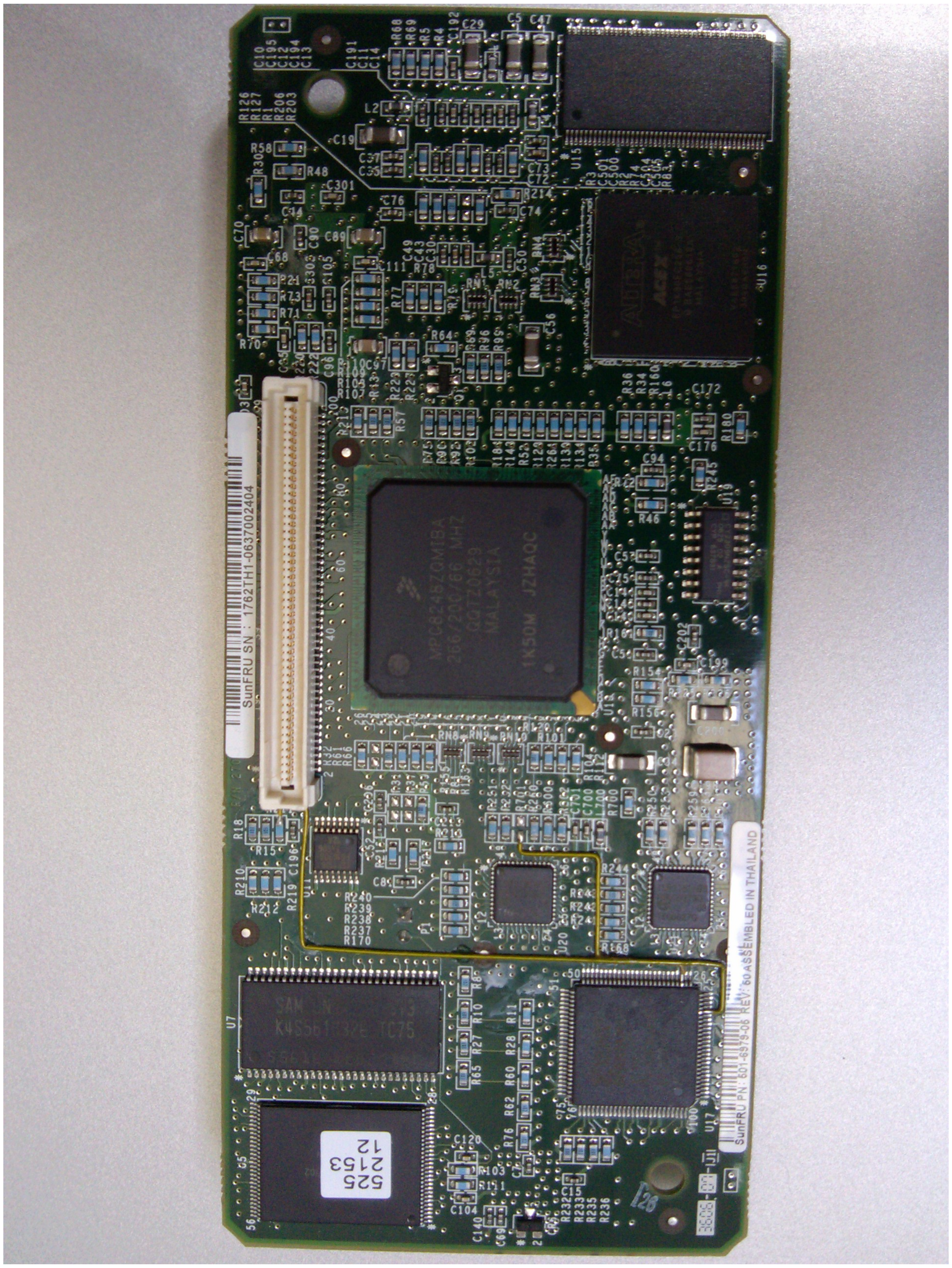


Figure 3.2: An ILOM from a Sun Fire X4200 M2 server.

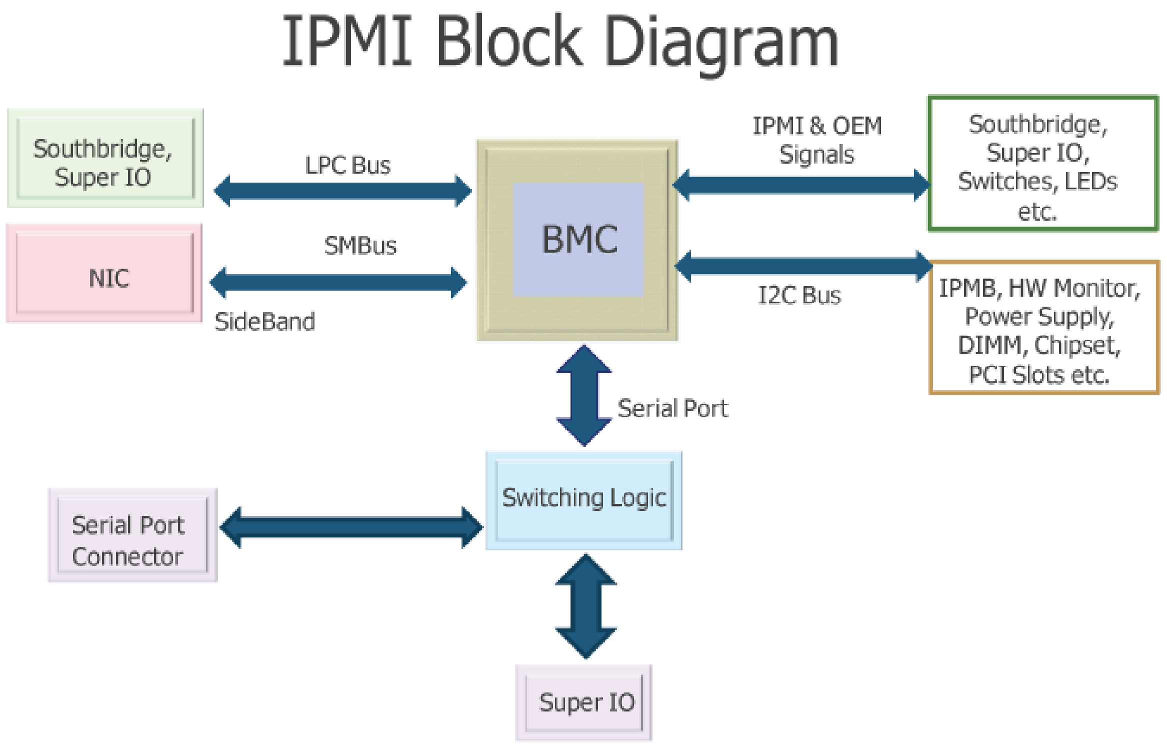


Figure 3.3: IPMI Block Diagram [62], illustrating the Baseboard Management Controller as the center of an IPMI system.

- Son of Grid Engine (SGE) [60], another open source offering based on the last open source release of Sun's SGE.

- Univa Grid Engine [56], a commercial offering, based on the last open source release of SGE. Univa continues to maintain an open source core offering [11], and employs many of the original Sun Grid Engine engineers [55].

For a job scheduler to work it must know the characteristics of the available hardware, which is usually determined during job scheduler installation and configuration. A batch job scheduler can be configured several different ways depending on the preferences of the system administrator or the requirements given to the system administrator by researchers or management. In a very basic configuration, users submit jobs into the HPC environment and the job scheduler queues the jobs up, running them in order as resources become available.

In more advanced configurations, custom job queues can be created with some having higher priority than others, allowing lower priority jobs to be temporarily suspended while higher priority jobs run. Custom job queues can also be used to partition up where jobs run. Custom job queues can be used such that certain researchers or research groups are only allowed to run on certain nodes. This is appealing if the HPC hardware is not homogeneous, so that researchers doing resource intensive simulations don't inadvertently run their jobs on inadequate hardware and cause their job to either take too long to run, or to fail due to it consuming all the resources and being killed due to an OOM condition [31]. Conversely, this would also keep other researchers from running jobs on these high end nodes. Schedulers can also be used to ensure that certain researchers or research groups can only consume a specified level of resources. These are just a few examples of what a job scheduler can do.

Aside from queuing and starting jobs, and managing priorities and where jobs run, a job scheduler also serves as a load balancer, trying to ensure that an HPC environment stays as close to 100% utilization as possible. Similar to how a restaurant host trying to optimize the number of people being seated, may move a party of two ahead of a party of four, a job scheduler will compare the available system resources to the requested resources of the jobs in a queue, and may move certain jobs ahead of others accordingly.

A job scheduler is also beneficial to the user. The user does not have to manually decide where a job needs to run, but simply states "I have this type of job that needs these many processors and at least this much memory", and the scheduler takes care of the rest. Additionally, it can email the user when their job starts, if the jobs state changes, and when the job completes. The use of a job scheduler is a necessity in an HPC environment and a winning move for both the system administrator and the user. Job schedulers are also capable of doing process accounting, which managers find appealing.

Servers in an HPC environment are commonly referred to as "compute nodes", or just "nodes". In this thesis, the term "node" and "server" are used interchangeably. The job scheduler in use for the environment in this paper is Sun Grid Engine version 6.1u4

3.3 Databases

3.3.1 Round Robin Database

Round Robin Databases, or RRDs, are exceptionally well suited for storing time series data. Round Robin Databases are those created by RRDtool (Round Robin Database Tool) [44, 67]. In addition to being used in this project, it is used in many other projects, including three of the projects cited in Section 2.1: Ganglia, Cacti, and Munin. The structure of an RRD is quite unique for people coming from traditional database backgrounds, so we'll cover it in a little bit of detail here.

An example RRD could be created on the command line as shown in Figure 3.4. A breakdown of the command is as follows:

```
rrdtool create \
  hostname.rrd \
  --start $(date +%s) \
  --step 60 \
  DS:temp:GAUGE:120:U:U \
  RRA:AVERAGE:0.5:2:720 \
  RRA:AVERAGE:0.5:5:8928
```

Figure 3.4: A example to illustrate the creation of databases with RRDtool

- *hostname.rrd*: this is the database name.
- *--start $(date +%s):* this is the start time of the database, declared in Unix time [57].
- *–step 60:* this how often (in seconds) the RRD should expect new data points.
- *DS:temp:GAUGE:120:U:U:* this is the data source (DS) declarations. This line is a set of parameters itself:

  - *DS:* this is a keyword, declaring a data source
  - *temp:* this is the name of the data source. This is needed because each RRD can have multiple data sources.
  - *GAUGE:* this is the type of data source being used. We are using the "GAUGE" type here, as the values we are reading can be thought of as the type read from a temperature gauge. There are other data source types that can be used: COUNTER, DERIVE, ABSOLUTE, and COMPUTE. More details on these types can be found in the RRDtool documentation [46].
  - *120:* this is how long, in seconds, the RRD database can go without receiving a data point. An "UNKNOWN" value is inserted into the RRD if the heartbeat value is exceeded without receiving a data point. The theory behind this feature is outlined in the RRDtool documentation.
  - *U:* this field specifies a minimum value. Here we place a "U", as we don't care to declare a minimum value. If we do, and any data point less than this is recorded, it will be recorded as "UNKNOWN".
  - *U:* this field specifies a maximum value. Here we place a "U", as we don't care to declare a maximum value. If we do, and any data point greater than this is recorded, it will be recorded as "UNKNOWN".

- *RRA:AVERAGE:0.5:2:720:* this is the Round Robin Archive (RRA) declarations. This line is a set of parameters itself:

  - *RRA:* this is a keyword, declaring a round robin archive

  - *AVERAGE:* this is the type of consolidation function (CF) being used. A CF is used to merge multiple data points, into a “consolidated data point”. There are currently four different types of consolidation functions: AVERAGE, MIN, MAX, and LAST. We are using the “AVERAGE” CF, which is self explanatory. More details on the other consolidation functions can be found in the RRDtool documentation.
  - *0.5:* this is the xff (xfiles factor), which states what portion of the consolidation function interval may be made up from “UNKNOWN” data. Values can range from 0 to 1.
  - *2:* this is how many data points are to be used by a CF for the creation of a consolidated data point.
  - 720: this is how many consolidated data points make up an RRA.

- *RRA:AVERAGE:0.5:5:8928:* this is another Round Robin Archive (RRA) declaration. The only difference here is the last two parameters:
  - 5: for this RRA, we’re using 5 data points for the creation of a consolidated data point
  - 8928: here 8928 consolidated data points are making up the RRA

That may seem like a lot, but it is quite straightforward really. The above command, in Figure 3.4, will create an RRD at start time “now”, and expect data points every 60 seconds. It will be looking for one data point named “temp”, of type “GAUGE”. Every two data points it will AVERAGE, putting them into an RRA. The RRA is capable of holding 720 data points.

Since each data point is read every 60 seconds, and each consolidated data point is made up of 2 data points, then we can determine that this RRA represents 24 hours (60 * 2 * 720 = 86400 seconds). To help illustrate the concept of an RRA and a CF, see Figure 3.5.

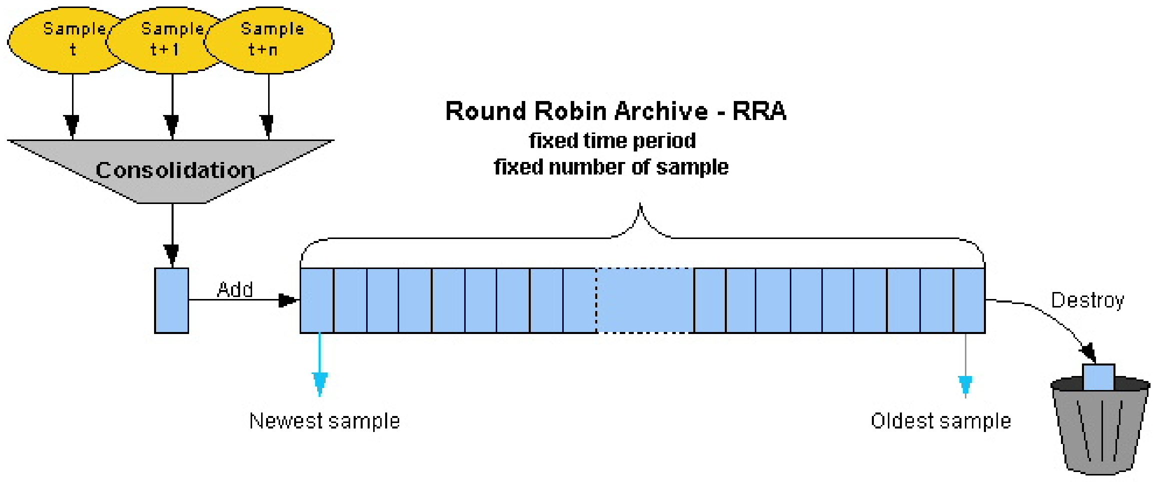


Figure 3.5: Concept of a Round Robin Archive [14]

Notice that there was a second RRA in the above command, in Figure 3.4. A database created by RRDtool can have one or more Round Robin Archives, which can be thought of as a fixed length queue or a circular buffer. Additionally an RRD can have multiple data sources. This concept is illustrated in Figure 3.6. Following the logic used in the first RRA, we can see that the second RRA in our example represents 31 days (60 * 5 * 8924 = 2678400 seconds).

Once the RRD is created, RRDtool can then update the database every *step*. The database is updated with time/data point pairs. In our example where a data point is every 60 seconds, RRDtool's update command could be called every 60 seconds. Alternatively, time/data point pairs can be buffered (programmatically or otherwise) and written to the database at a later point. The database doesn't care what time its

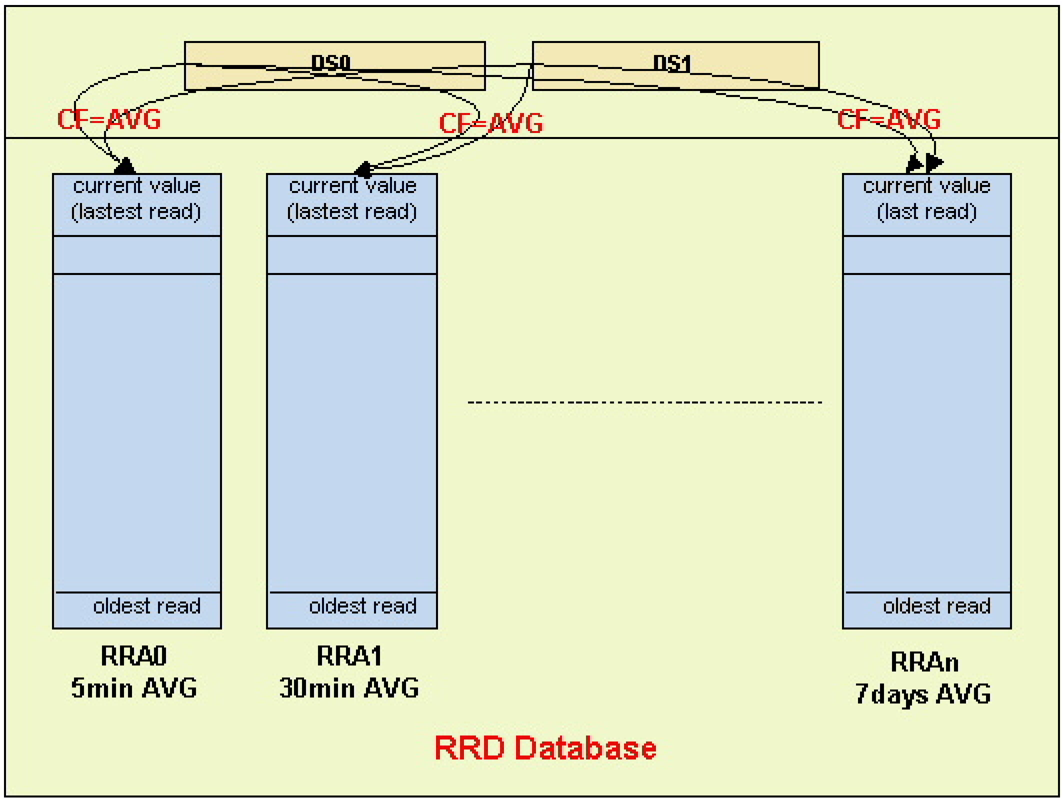


Figure 3.6: An RRD with multiple DS and multiple RRAs [24]

values are actually updated, since the user is providing a timestamp with each data point.

At any point after the database is created, RRDtool's graph command can be called. At a most basic level, RRDtool's graph function will allow you to graph the data that you've recorded. It must be provided with a few options, such as "start" and "end" time. An example would be an end time of "now", and a start time of an hour ago. This results in a graph period of one hour, using the most recent data. The graph function could also be asked to graph something like from 8:00am yesterday morning to 8:00am this morning. That would be a much larger graph window, using data that is a little older. This is where having RRAs of varying precision and size becomes useful. RRDtool will intelligently choose the best RRA, based off of the requirements of the requested graph.

RRDtool also provides the ability to graph multiple RRDs and multiple DS to one graph, see Figure 3.7.

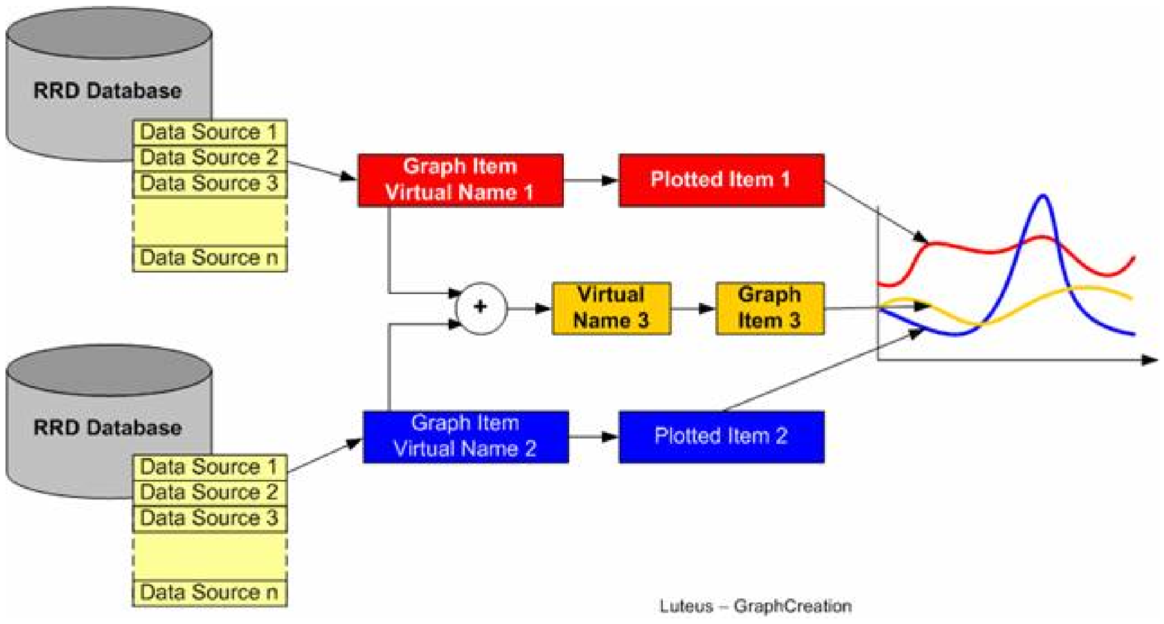
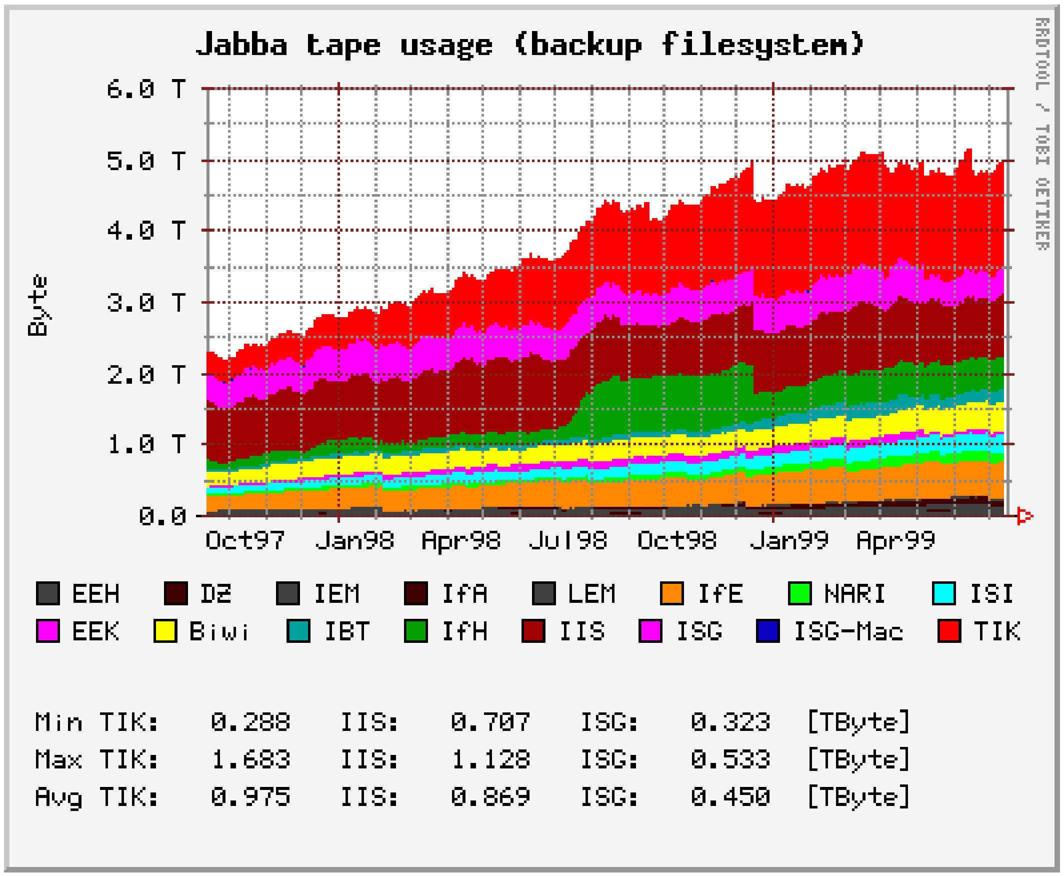


Figure 3.7: A graph comprised of multiple DS and multiple RRDs [10].

Additionally, you can run data points through a function to get a new data point. As an example, you could convert bits to bytes by multiplying by eight, or even create a plot that was a function of other plots. RRDtool's graphing ability is too complex to list here, but more details can be found in the RRDtool documentation. An example of a graph created with multiple data sources can be seen here in Figure 3.8.



```
Min TIK:   0.288   IIS:   0.707   ISG:   0.323  [TByte]
Max TIK:   1.683   IIS:   1.128   ISG:   0.533  [TByte]
Avg TIK:   0.975   IIS:   0.869   ISG:   0.450  [TByte]
```

Figure 3.8: A graph created with multiple DS [7].

3.3.2 Relational Database

Relational Databases [72, 68], formally implemented in a Relational Database Management System (RDBMS), are often loosely referred to as an SQL (Structured Query Language) Database. Technically, SQL is the programming language used to communicate with an RDBMS.

SQL itself is an ISO standard [17], that like many programming languages has seen revisions over the years. There are many SQL database implementations such as Microsoft SQL Server [21], Oracle Database [36], MySQL [28], PostgreSQL [41], and SQLite [51]. Many of these implementations add their own extensions to the SQL language.

SQLite is unique among the other four databases mentioned above in that it isn't a standalone RDBMS, but rather an embedded RDBMS. That is, instead communicating with another process (program) either on the same machine or over the network, SQLite will be the same process, as it will be embedded into the application. SQLite is used in millions of applications - from iPhones, Mac OS X, Mozilla Firefox, and many others - making it possibly the most used SQL database in the world [52].

The fundamental concept behind a relational database is that tables of data can be linked together (related to) with a "key". A data field in one table can act as the key to one or more rows of data in another table. An example of this can be seen in Figure 3.9.

Many different types of relational models can exist. Some common models are "one-to-one", "one-to-many", and "many-to-many". In a one-to-one relationship, there will be one row in a table corresponding to one row in a foreign table. An example could be a person and a social security number. One person should have exactly one social security number, and a social security number should belong to only one person. An example from the IT world could be users and profiles, each user having exactly one profile. A diagram of a one-to-one relationship can be seen in Figure 3.10.

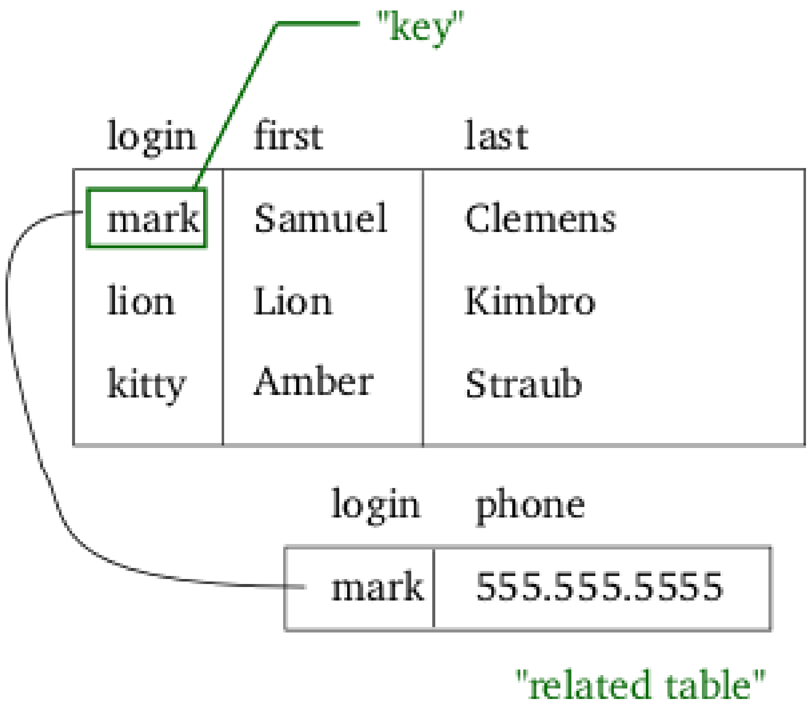


Figure 3.9: An example of a key in one table referencing data in another table [61].

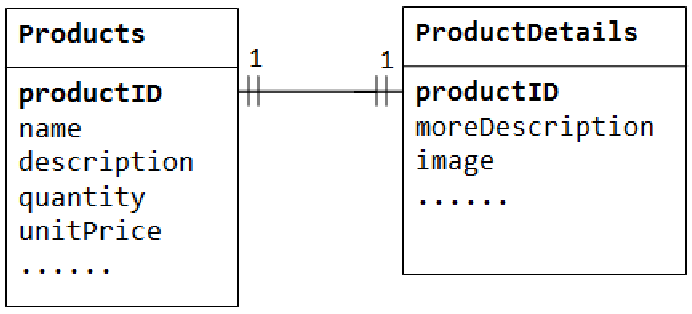


Figure 3.10: An example of a “one-to-one” relationship between Products and Product Details. Modified from [43].

In a “one-to-many” relationship, there will be one row in a table corresponding to one or more rows in a foreign table. An example could be person and phone numbers. Most people these days have multiple phone numbers - work, home, mobile, and Google Voice. Another example could be a person and email addresses, as many people have multiple email address. They may have a primary personal email address, a throwaway email address for mailing lists, and a work email address. Both of these examples illustrate a one-to-many relationship. A diagram of a one-to-many relationship, in this case Teachers and Classes (a Teacher can have one or more classes), is illustrated in Figure 3.11.

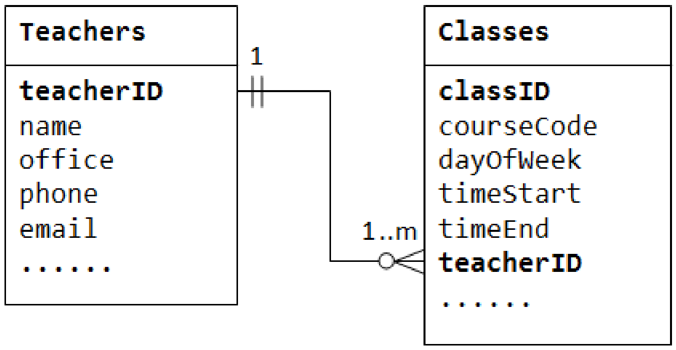


Figure 3.11: An example of a “one-to-many” relationship between Teachers and Classes. Modified from [43].

In a “many-to-many” relationship, there will be one row in a table corresponding to one or more rows in a foreign table. Conversely, the foreign table could include a row that corresponds to one or more rows in the original table. Anyone who has ever used Gmail and “labeled” their emails should immediately understand this. In this situation, there would be a tabled called “messages” and a table called “labels”. A message could have zero, one or more labels applied, and the same label could appear on one or more messages. An example can be seen in Figure 3.12.

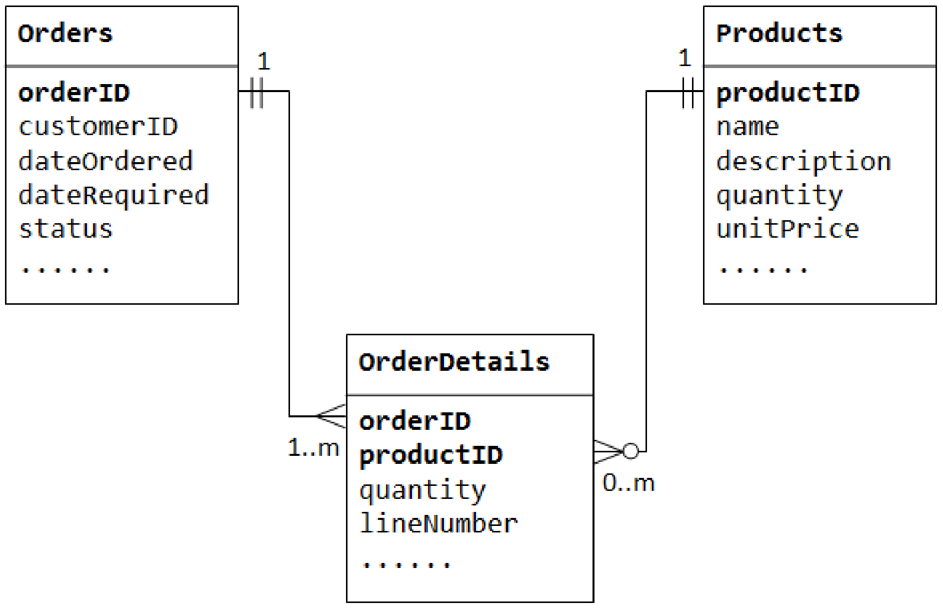


Figure 3.12: An example of a "many-to-many" relationship between Orders and Products [43]. Here a junction table [18] is used to support the relationship.

The database used in this project is SQLite. If desired, the SQL implementation could be changed in minutes with no code change required. This will be covered in Section 4.4.

3.4 Putting it together

Our idea revolves around the idea of tying the IPMI systems into a larger system, giving us their included temperature monitoring capabilities on a localized level, but with a global perspective. Local temperature events can be monitored, and responded to, while at the same time the larger system would be smart enough to realize if this is happening on a larger scale, allowing it to act at a global level, before waiting for each individual system to trigger a temperature event.

The IPMI system gives us further control over the system, allowing us to shut down the system by turning off the power, if need be. The scheduler gives us the ability to suspend and restart jobs to manage power events.

If we had a global temperature event, such as losing one of the A/C compressors, and the temperature in the server room began to rise, pushing the ambient temperature above where it should be, the larger system could respond by first telling the scheduler to suspend (pause) all jobs running on the HPC cluster, and to shut down all compute nodes not running any jobs. If this caused, say, half of the compute nodes to turn off, and caused the other compute nodes to drop their temperature output by some significant percentage, that would very possibly be manageable by the remaining A/C capacity. If the temperature continued to rise, additional steps could be taken to curb temperature output while still trying to find a balance between both saving long running compute jobs and hardware. After the HVAC techs repair the A/C compressor, the system administrator can resume the paused jobs and power the compute nodes in question back on.

The following chapter covers the implementation of our system. An overview of the architecture can be seen in Figure 3.13.

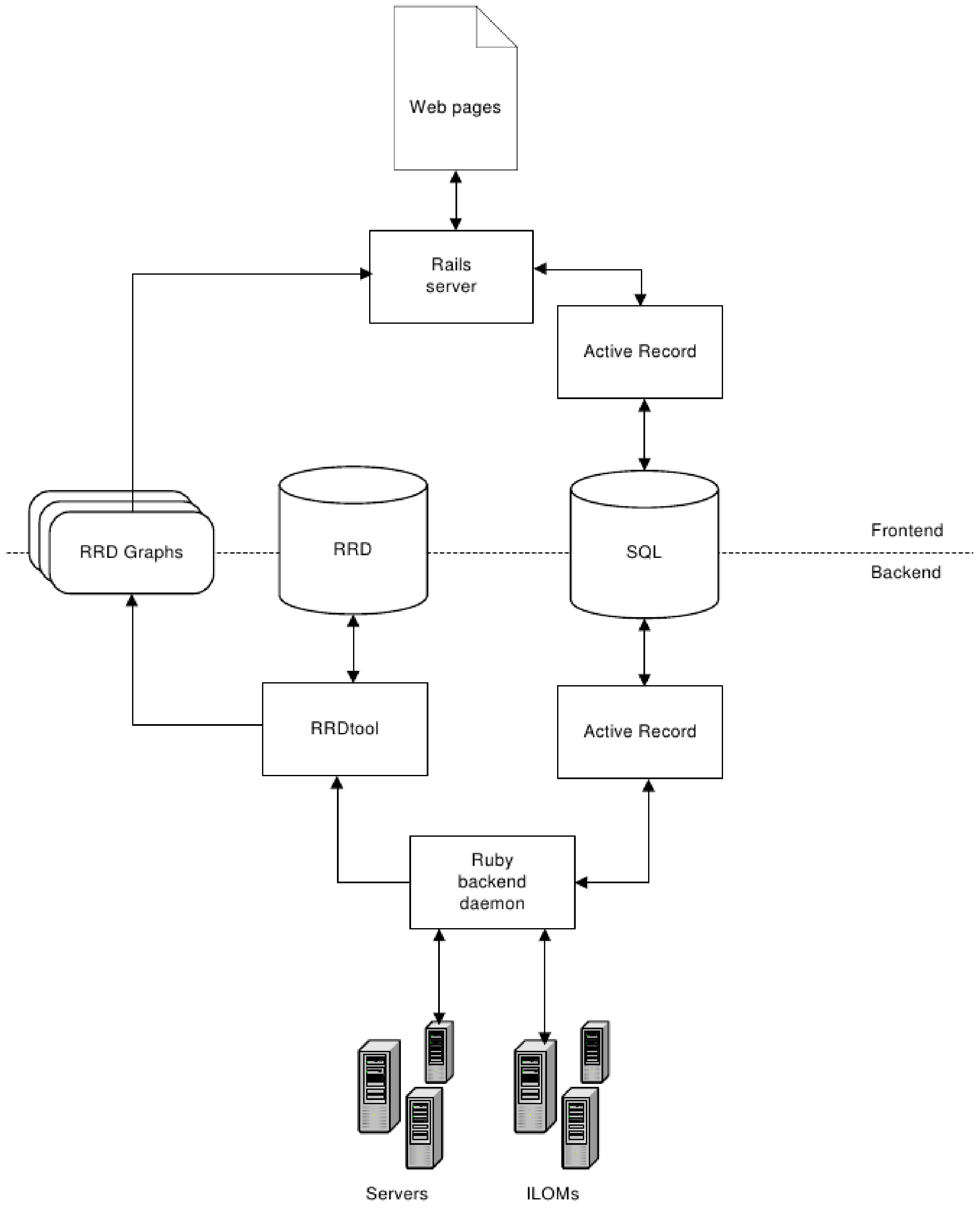


Figure 3.13: System Architecture

Chapter 4

Implementation

The implementation of this project occurred in stages. First we had to find the best way to talk to the ILOMs. OpenIPMI [33] has Python language bindings, so early testing leveraged those bindings. We quickly discovered that OpenIPMI didn't work quite the way we wanted, not to mention that there were other issues with the Python bindings for OpenIPMI, because they were created with SWIG [50], so other options were explored.

We knew that we could get exactly the information we wanted from IPMItool [16], a standalone program. So why not wrap calls to IPMItool in another language? We knew that there would eventually be a web based frontend to this program, and Rails [47, 69] was a desirable choice on a number of levels, so the decision was made to use Ruby [48, 64] all the way through the development process.

From a high level, the project consists of two components:

- The first component is a backend daemon that runs in the background, polling the temperature sensors across all of the servers every minute, and writing the values to two types of databases. An SQL database keeps all of the *Node* (server and ILOM) data, as well as *Location* data (which server rack, and where in that rack). RRD, or round robin databases are exceptionally well suited for time series data, which in this case, are the temperature values. This allows us to do a number of things, such as graphing a window of recent temperature values, or going back to recreate graphs from weeks ago, or create a large graph comprising of the last month. The backend daemon is also responsible for taking specified actions when read temperature values exceed defined thresholds.

- The second component is a Rails based frontend. The frontend has no knowledge of the backend program, nor does the backend have knowledge of the frontend. The frontend program is merely a window into the databases. Additionally, the frontend program provides the ability to manipulate some of the data in the databases.

4.1 Ilom class

The initial Ruby backend for communicating with the ILOMs simply wrapped calls to IPMItool, and worked fine. However, it was quickly realized that calls to the ILOMs would be faster if we could query a specific sensor, and not all of the sensors, or all of a class of sensors, such as temperature, and then parse through the output, as we had been doing in this initial version. IPMItool didn't offer this capability, allowing us to query a sensor by its ID number, but it was discovered that another suite of tools for talking to IPMI did: FreeIPMI [8].

FreeIPMI was designed with HPC environments and clusters in mind, and while there is significant overlap with IPMItool, its feature set and capabilities are distinctly different.

Unfortunately, FreeIPMI doesn't have Ruby bindings, so we again had to wrap calls to another command line program. The FreeIPMI program provides the command line tool "ipmi-sensors" [20] for talking to sensors which is what we wrapped.

Only the portion of the Ilom Ruby class that talks to sensors was replaced with code that wraps "ipmi-sensors". IPMItool is still used in other parts of the ILOM class, as it is actually a more mature product.

The Ilom Ruby class object grabs a bunch of information upon being initialized, as illustrated by Figure 4.1.

```
  def initialize(hostname = "localhost")
    @hostname = hostname
    @ipmi_authcap = check_authcap_needed
    @ipmi_endianseq = check_endianseq_needed
    @mbt_amb,@mbt_amb_id = find_temperature_device
    @cur_temp = read_cur_temp
    @temp_status = read_cur_temp_status
    @thresh_unc,@thresh_ucr,@thresh_unr = read_temp_thresholds
    @watermark_high = @watermark_low = get_temp
  end
```

Figure 4.1: An Ilom class object

A quick walkthrough of the class instance variables are as follows:

@hostname is the hostname of the IPMI device.

@ipmi_authcap, and *@ipmi_endianseq* are workaround flags for FreeIPMI to communicate with certain types of IPMI devices. Setting them in the object class eliminates the need to recheck for them every time. If the instance variable is set, we just send that workaround along with the FreeIPMI command.

The interesting part here is that we have to use IPMItool to talk to the device initially to determine what type of device it is, and whether a workaround flag is needed for FreeIPMI.

@mbt_amb, and *@mbt_amb_id* are the name of, and numeric id of, the ambient temperature sensor. We store these values so that in the future, we can ask things like "what is the temperature of *@mbt_amb_id*".

@cur_temp is the current temperature, as last read from the ambient temperature sensor.

@temp_status is the temperature status returned by the IPMI device. When the temperature is in a safe range, it will be an "OK" string.

@thresh_unc, *@thresh_ucr*, and *@thresh_unr* are the "Upper Non Critical", "Upper Critical", and "Upper Non Recoverable" thresholds, respectively.

An interesting point here is that some platforms don't define all three of these values. Some omit the "Upper Non Recoverable" value. Other IPMI devices don't define any of them.

@watermark_high, and *@watermark_low* are historic - over the life of the class object - high and low temperature values. During class initialization, they won't be any different than what the initial temperature reading is.

The ILOM class also includes other functionality such as querying the chassis power state and being able to turn the server on an off. For this other functionality we didn't feel there was a need to have a class instance variables defined.

4.2 IlomRRD class

This class sits on top of the official RRDtool Ruby bindings [45]. This class serves two main purposes.

First, the class creates and updates RRD (Round Robin Databases) every minute when new temperature values are polled. These values are written to an RRD for that particular server. Each server has its own RRD. Figures 4.2 and 4.3 show the class actions for creating and updating an RRD. It is clear that the syntax for the Ruby class is no different than that of RRDtool on the command line.

Secondly, the class knows how to create graphs from the RRDs. The graphs are updated every five minutes. The graphs include hostname, current ambient temperature, as well as temperature thresholds. Time values correspond with the X axis, while temperature values correspond to the Y axis.

```
def create
  RRD.create(
    @name,
    "--start",
    "#{@start_time - 1}",
    "--step",
    "#{@step}",
    "#{@ds}",
    "#{@rra1}",
    "#{@rra2}"
  )
end
```

Figure 4.2: Creating an RRD with RRDtool's Ruby Bindings

```
def update v
  RRD.update(
    @name,
    "N:#{v}"
  )
end
```

Figure 4.3: Updating an RRD with RRDtool's Ruby Bindings

Values required for RRDtool create and update the RRDs, as well as graph the temperature values are read from a YAML [53] configuration file. Values such as database size has to be declared upon creation. RRD graph creation has many options such as line type and color, and temperature viewing window (e.g., 1 hour of temperature data). If we wanted to change any of these parameters, it would be as simple as changing the YAML configuration file.

The IlomRRD class assigns several values upon initialization, as shown in Figure 4.4.

```
def initialize(h)
  @name = h[:db_path] + "/" + h[:name] + ".rrd"
  @r_db_path = (h[:db_path] + "/").reverse
  @graphname = h[:img_path] + "/" + h[:name] + ".png"
  @graphname_sm = h[:img_path_sm] + "/" + h[:name] + ".png"
  @start_time = Time.now.to_i
  @step = h[:step]
  @ds_label = h[:ds_label]
  @ds_type = h[:ds_type]
  @ds_heartbeat = h[:ds_heartbeat]
  @ds_min = h[:min]
  @ds_max = h[:max]
  @ds = "DS:#{@ds_label}:#{@ds_type}:" + \
      "#{@ds_heartbeat}:#{@ds_min}:#{@ds_max}"
  @rra1 = "RRA:#{h[:rra1][:cf]}:" + \
      "#{h[:rra1][:xff]}:#{h[:rra1][:step]}:#{h[:rra1][:rows]}"
  @rra2 = "RRA:#{h[:rra2][:cf]}:" + \
      "#{h[:rra2][:xff]}:#{h[:rra2][:step]}:#{h[:rra2][:rows]}"
  @rra_cf = "#{h[:rra1][:cf]}"
end
```

Figure 4.4: An IlomRRD class object

4.3 IlomSGE class

This class sits on top of the DRMAA Ruby bindings [65]. We wanted to leverage DRMAA (Distributed Resource Management Application API) [30] as much as possible for this class, as DRMAA provides a standardized way to communicate with any scheduler that supports DRMAA, such as SGE.

The DRMAA Ruby bindings had been created and hosted by Sun, but after the Oracle acquisition of Sun they disappeared as Oracle folded much of Sun's site into their own. After a little research, we found that the DRMAA 4 Ruby code had made its way onto github, updated to work with Ruby 1.9.

An issue that preventing us using DRMAA exclusively is that there appears to be no way to query all of the running jobs using DRMAA. After searching through a number of DRMAA documentation and mailing lists, this seemed to be confirmed [70].

To get around this, we had to wrap the output of the command line SGE program *qstat*, to get a list of running jobs. We ask *qstat* to return the job list in XML format. We leverage the *nokogiri* Ruby gem to parse the XML output to build an array containing all of the jobs.

Our remaining class methods leverage the DRMAA interface, allowing us to suspend, resume, and terminate jobs. With DRMAAv2, however, wrapping *qstat* as a workaround should no longer be required [71], allowing a class using only the DRMAA interface.

4.4 Database backend

The backend consists of an RRD for storing time series data (temperature values). As well as SQL databases for recording server characteristics, server location data, server status, and current temperature values.

The Ruby ActiveRecord [2] gem is used to leverage the SQL database. ActiveRecord allows you to interface with the SQL database without writing any SQL code (although you can). Should we want to change the database at any time, ActiveRecord makes this incredibly easy, as it is database agnostic. No code change is required, only a configuration file change is needed.

There are two main Ruby models with which we are interacting. A Node model as seen in Figure 4.5, and a Location model, as seen in Figure 4.6.

```
Node(

    id: integer
    hostname: string
    cur_temp: float
    temp_status: string
    thresh_unc: float
    thresh_ucr: float
    thresh_unr: string
    watermark_high: float
    watermark_low: float
    created_at: datetime
    updated_at: datetime
    ipmi_authcap: string
    mbt_amb: string
    mbt_amb_id: string
    def_thresh_unc: float
    def_thresh_ucr: float
    def_thresh_unr: string
    unc_exceeded: boolean
    ucr_exceeded: boolean
    unc_exceeded_counter: integer
    ucr_exceeded_counter: integer
    ilom_name: string,
    master: boolean
    )
```

Figure 4.5: The structure of the Node model

You'll notice that many of the values in the Node model may look familiar from the ILOM class, and they should be, many of them are the same, However for completeness, we'll cover them again here. An overview of the values is as follows:

- *id* is the database id.
- *hostname* is the server hostname. Note that in the ILOM class, it is the ILOM hostname that is stored, but the value stored here is the server hostname.

```
Location(

    id: integer
    rack: integer
    rank: integer
    hostname: string
    created_at: datetime
    updated_at: datetime
    node_id: integer
    )
```

Figure 4.6: The structure of the Location model

- *cur_temp* is the the current temperature, as last read by the ambient temperature sensor

- *temp_status* is the temperature status returned by the IPMI device

- *thresh_unc, thresh_ucr,* and *thresh_unr* are the “Upper Non Critical”, “Upper Critical”, and “Upper Non Recoverable” thresholds respectively. Note that *thresh_unr* is a string, whereas the other two are floats. This is because some IPMI implementations don’t define an “Upper Non Recoverable” value. Making it a string, we can insert “N/A” into that database field in that case.

- *watermark_high,* and *watermark_low* are historical high and low temperature values

- *created_at,* and *updated_at* are values generated automatically by ActiveRecord.

- *ipmi_authcap* is a workaround flag for the ILOM class, specifically for FreeIPMI

- *mbt_amb,* and *mbt_amb_id* are the name of and numeric id of the ambient temperature sensor

- *def_thresh_unc, def_thresh_ucr,* and *def_thresh_unr* are the “default” “Upper Non Critical”, “Upper Critical”, and “Upper Non Recoverable” values as read from the ILOM originally. This allows us to go back the original values should we override the values from the web frontend, and later realize we want to go back to the defaults.

- *unc_exceeded,* and *ucr_exceeded* are flags that are set if the “Upper Non Critical” or “Upper Critical” thresholds are exceeded.

- *unc_exceeded_counter,* and *ucr_exceeded_counter* are used to record how much time has been spent above the “Upper Non Critical” and “Upper Critical” thresholds, respectively.

- *ilom_name* is the hostname of the IPMI device associated with the server.

- *master* is a Boolean variable specifying if the node is the “master” node. This will commonly be the head *node* or backend *node(s)* of the system, and the *node* this software is running on, should it be a different *node.*

The Location model is used exclusively by the web frontend. It contains a rack and rank value, which are server room coordinates for which rack, and what slot in the server rack. This Location model is used as a mapping for the Node model. A quick overview of the values is as follows:

- *id* is the database id

- *rack* is which rack the server resides in

- *rank* is the position, from the top of the rack, of the server

- *hostname* is the server hostname
- *created_at,* and *updated_at* are values generated automatically by ActiveRecord
- *node_id* is the foreign key id for Node. It is the mapping that allows a Location to find its corresponding Node.

4.5 Rails frontend

The system frontend is illustrated by Figure 4.7.

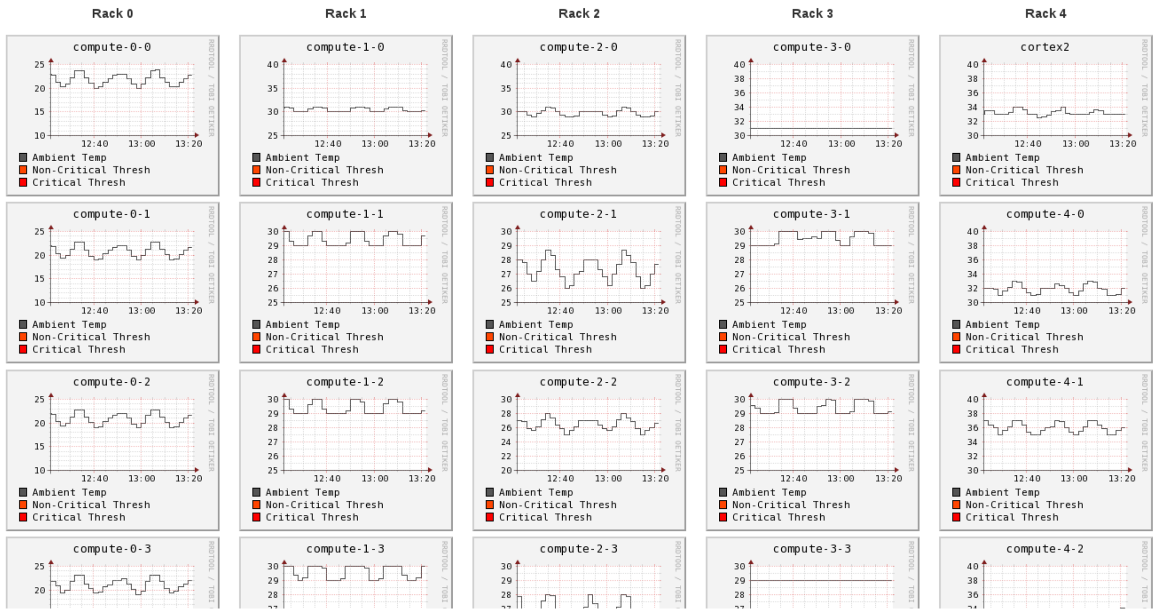


Figure 4.7: The default frontend view, RRDtool generated temperature graphs of all of the nodes. The layout of the graphs on the webpage match the physical layout of servers in the server room. Clicking on any of the graphs will bring up a larger graph, along with other characteristics of the server, pulled from the SQL database. Some of these values, such as the temperature thresholds, can be overridden and written back to the SQL database from here.

The frontend is primarily designed to give the system administrator a visual representation of what is going on, as well as giving the system administrator the ability to override some of the database values. For instance, the ILOM may report the an upper temperature threshold of 70 degrees. The system administrator could, from this web interface, override that value dropping it to 50 degrees. If a threshold value isn't defined by an IPMI based device, that value can be specified here.

4.6 Logic/Algorithm

Upon startup, several configuration files are read.

The first configuration file, *thresholds.yml* is read. This is where actionable values are read from should there be a thermal event. The second configuration file, *layout.yml*, is where the *Location* model gets it mapping from. Should a server be moved, just edit this file, reload the backend, and the server will appear in the correct location in the web interface. Next, the RRD configuration file, *rrd_config.yml*, is read, providing parameters for the configuration of the RRD database, and parameters for RRD graphs. The file, *hosts.txt* is read next, which is just a pairing between hostnames and ILOM hostnames. The backend program uses this to determine which ILOMs belong to which servers. The pairings from this file are loaded into a hash named *hostlist.* One last file, *master.txt* is read, which the backend will use to assign the status of "master" to one server.

At this point, the backend program loops through every listing in the *hostlist*, checking to see if it is already known - that is, if it already exists in the SQL database. If it is known, it moves on to the next value. If it isn't known, it creates a *Node* object and creates an *Ilom* object. The *Ilom* object initialization process polls its associated ILOM acquiring temperature values, thresholds, and sensor IDs. Corresponding values are then assigned to the *Node* object which are then written to the SQL database. The code for this process is shown in Figure 4.8.

```
hostlist.each_key do |x|
  rrd[:name] = x
  nodelist[x] = { :bmc => Ilom.new(hostlist[x]), :rrd => IlomRRD.new(rrd) }
  # a quick mapping because the Node model doesn't
  # consider the Ilom to be a separate entity and the Ruby classes do.
  if !( n = Node.find_by_hostname(x) )
    t = Node.create( { :hostname => x } )
    nodelist[x][:node_id] = t.id
    t.master = (x == master_node ? true : false)
    t.hostname = x
    t.ilom_name = hostlist[x]
    t.watermark_low = t.watermark_high = t.cur_temp = \
        nodelist[x][:bmc].get_temp
    t.temp_status = nodelist[x][:bmc].get_temp_status
    t.thresh_unc, t.thresh_ucr, t.thresh_unr = \
        nodelist[x][:bmc].get_temp_thresholds
    t.thresh_unr = t.thresh_unr[0,4] #trim (thresh_unr is a string, ATM)
    t.def_thresh_unc, t.def_thresh_ucr, t.def_thresh_unr = \
        t.thresh_unc, t.thresh_ucr, t.thresh_unr
    t.mbt_amb = nodelist[x][:bmc].get_temp_device t.mb
    t_amb_id = nodelist[x][:bmc].get_temp_id
    t.unc_exceeded = false
    t.ucr_exceeded = false
    t.unc_exceeded_counter = 0
    t.ucr_exceeded_counter = 0
    t.location = Location.find_by_hostname(x)
    t.save
  else
    nodelist[x][:node_id] = n.id
  end
end
```

Figure 4.8: Node creation

During this loop a "*nodelist*" is constructed, and also given a reference to its corresponding *Node.* This *nodelist*, which is a nested hash, will be used by the backend program will use for the remainder of its life. The structure of the *nodelist* can be seen in Figure 4.9:

```
    {hostname => { :bmc => Ilom object ,
                   :rrd => IlomRRD object,
                   :node_id => Node ID
                  }
    }
```

Figure 4.9: The structure of a *nodelist* entry

So each entry in the *nodelist* can reference its Ilom object information, its Ilom-RRD object information, and its Node ID for database access.

After it is verified that every host has an entry in the SQL database, or one created if it didn't previously exist, the backend program moves on to initializing RRD databases. At this point RRD databases are considered expendable, an no effort is made to preserve the previous contents if an RRD database for a given server already exists. They are simply reinitialized per the values read from *rrd_config.yml* and the backend program continues.

4.6.1 Main loop

At this point the main loop is entered, which is where the program will live until terminated or restarted. The main loop only runs once a minute, so once the loop completes, it sleeps until the minute is up, at which point it wakes up and runs again. This is accomplished by reading the time at the beginning of the loop, and then looking at the time that has elapsed when execution reaches the end of the loop. Sleep is then called for 60 seconds, less whatever the delta of elapsed time is.

The general logic of the main loop is as follows:

Loop through each entry in the *nodelist*, reading the temperature sensor value from that server, and then updating the RRD database for that server, as well as the *cur_temp* value in the SQL database.

If the *cur_temp* value is greater than *watermark_high* or lower than *watermark_low* then update those values in SQL as well.

Here we check to see if there is a thermal event occurring. To make the main loop easier to read, we'll cover this event logic separately.

If this is a fifth loop, update the RRD graphs.

A flowchart of the main loop, including the thermal event logic, can be seen in Figure 4.10.

4.6.2 Thermal Event Logic

The thermal event logic happens in multiple stages.

First we check to see if the critical threshold has been exceeded. If so, we set the global *temp_event* Boolean flag to true, and increment the *ucr_exceeded_counter* and set the *ucr_exceeded flag* to true for this particular server.

Then we compare the *ucr_exceeded_counter* with the *ucr_counter_thresh* read from the *thresholds.yml file* in the beginning. If the *ucr_exceeded_counter is* greater than or equal to the *ucr_counter_thresh* then we terminate all jobs on the server, and then shut down the node. To make this easier to read, shutdowns are covered separately

After this we do the exact same thing for non-critical thresholds.

We check to see if the non-critical threshold has been exceeded. If so, we set the

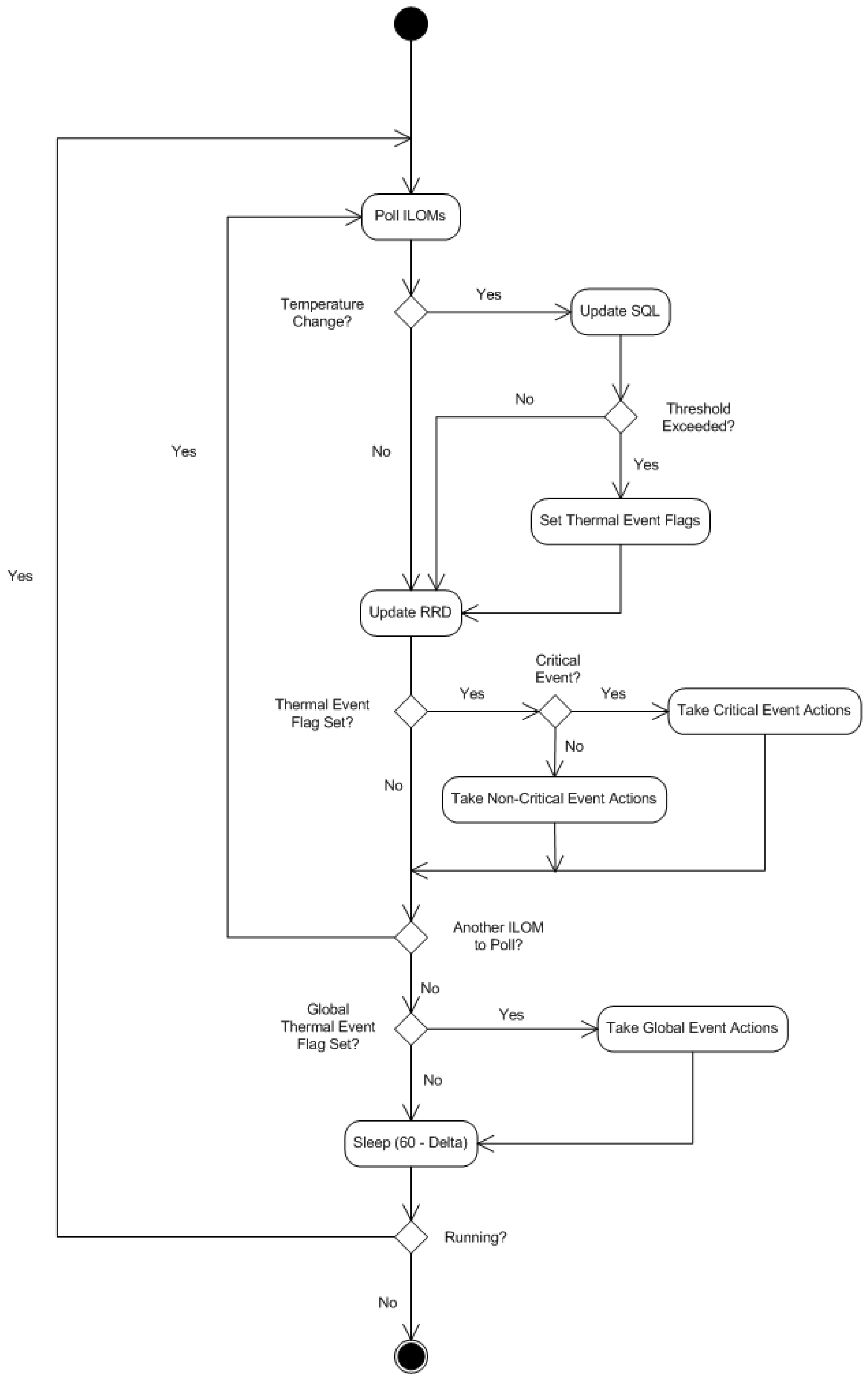


Figure 4.10: Flowchart of the main monitoring and control loop

global *temp_event* Boolean flag to true, and increment the *unc_exceeded_counter* and set the *unc_exceeded flag* to true for this particular server.

We then compare the *unc_exceeded_counter* with the *unc_counter_thresh.* If the *unc_exceeded_counter is* greater than or equal to the *unc_counter_thresh* then we terminate all jobs on the server, and then shut down the node.

Typically the non-critical thresholds can be thought of as a warning zone, but servers should not spend too much time there. As such, if we exceed the counter thresholds, the server will still be shut down. The default counter thresholds are three for non critical and one for critical. What this means is that the server can spend three event loops in the non-critical zone before it is shut down, but as soon as it enters the critical zone, it is shut down. This is just the logic for local events. After we check for local events, we look for global events.

After all of the servers in *nodelist* have been looped through, we check for global events. The default threshold for a global event is 5% of the total number of servers. This 5% value can be changed in the *thresholds.yml* file if the system administrator thinks it is too high or too low. The global logic is much simpler than the local logic, and works as follows:

If the global *temp_event* flag was set then we are going to search through the SQL database to find all nodes that have the *unc_exceeded* flag set.

If the number of nodes returned is less than 5% of the total number of nodes, it isn't consider a global event. The global *temp_event* flag will be set to false, and we won't check for another global event until another local event sets that flag again.

If the number the number of nodes returned is greater or equal to 5% of the total number of nodes, then the backend will believe it is dealing with a global event, and try to manage it.

Managing a global event involves three actions. First, all job queues are closed, so no further jobs can be submitted. Secondly, all running jobs across all nodes are suspended (not terminated as in the local event case). This will cause any computations to pause, bringing the temperature output down to that of an idling server. Third, all nodes that aren't running jobs will be shut down. These three actions are an attempt to drop the temperature output to something manageable, in the event that there was only a partial cooling failure. If this does stabilize the temperature output in the server room, months of computation time can be saved.

4.6.3 Shutdown methods

Shutdowns are attempted two different ways. First a clean shutdown will be attempted. The backend program has SSH keys to all the servers, and can issue a shutdown command via SSH. If it turns out the OS isn't shutting down in a timely manner, or SSH is unresponsive, then a hard shutdown command can be issued to the ILOM over IPMI. This hard shutdown method via IPMI has the same effect as walking up to the server and holding the power button for five seconds.

4.7 Execution

Currently the backend process is managed by the daemons [5] Ruby gem. The frontend, throughout the development process, has used the built in Ruby webserver, WEBrick [58]. For production, the frontend should move into being run by another webserver such as Apache [59]. Running a Rails based project on Apache can be accomplished with Phusion Passenger [38]. For the backend, daemons is suitable for production.

Chapter 5

Conclusion

Stepping back and looking at this project, we can see that we've filled a void not currently addressed by other monitoring solutions. Certainly we could have created a custom Nagios plugin to poll the IPMI system, and then write another Nagios event handler to take a predefined action such as shut down the node if it exceeds a temperature threshold, but beyond that, how far could that platform be leveraged? We've built a more robust platform here by not being limited to the constraints provided by trying to use an existing solution.

Currently we can detect and respond to local temperature events. Because we have a global perspective, each temperature event is handled not only as an individual event, but is also looked at in the context of the entire monitored environment to determine if a global temperature event may be occurring. If so, additional steps are taken to proactively curb the anticipated global rise in server room temperature. It is these proactive actions that give long running HPC jobs the greatest chance of surviving in the event of a partial cooling failure.

The web frontend is also an invaluable tool for the systems administrator, doubling as a visualization tool, making it easy to view temperature trends as the layout of the web interface mirrors the physical layout of the HPC environment. Additionally, we can override the temperature thresholds provided by the IPMI devices via the web frontend, view high and low temperature readings, as well as view historical temperature data.

This project has been tested throughout the development process. Each component has been tested adequately in a standalone capacity before being integrated with the rest of the code.

As everything was built from the ground up, we were able to let each iteration sit and run, to observe behaviors, see if it crashed, see if it did anything unexpected, before adding the next component.

The Ilom class is the oldest piece of code from this project, and has been running on HPC and non HPC servers for over six months. A lot has been learned about ILOMs and other IPMI based devices throughout this process, such as the fact that certain IPMI devices can and will arbitrarily return garbage values from time to time. We learn to handle these types of situations gracefully, so that we don't crash the system.

Since turning off the A/C in the server room wasn't a realistic way to test this project, nor was waiting for the next environmental failure, should it come, testing was achieved by manipulating the temperature thresholds. We tested this project by lowering the temperature thresholds at or below the ambient temperature on individual servers, attempting to create a local event on those servers. Both non critical and critical events were triggered in this manner. Global events were triggered in a similar manner, by lowering the temperature thresholds on 5% of the nodes, we were able to achieve a global event.

Based off of these actions, we are confident that should a real environmental failure occur, that not only will hardware be saved, but also long running HPC jobs.

Chapter 6

Future Work

We have identified numerous future directions for this project.

Parallelize The Main Loop

The first enhancement to this project would be to parallelize the main monitoring and control loop. In the current form of our project, scaling to installations larger than 150 nodes would be problematic. This is because the main loop operates sequentially. Right now, there are 136 nodes in our HPC installation, and roughly 150 compute nodes could be iterated over in our 60 second monitoring and control loop. The largest delay is querying the ILOM and waiting for a response. If we parallelize this main loop, this project could scale to HPC installations with thousands of nodes. The most straightforward way to do this may be to parallelize the the querying by server rack. The code base already knows about the physical layout of the server room, so no additional data would have to be supplied to the program. Each server in the rack would still be queried sequentially, but this would be happening in parallel across every rack in the installation.

For example, an installation with six server racks would spawn six threads where it steps through each server in a rack in parallel. This would largely guarantee that the loop completes in 60 seconds. A common server rack found in server rooms today is 42U (it could hold 42 1U servers, or 21 2U servers, etc.), so each rack in this example would have no more than 42 servers. In this thesis, we are currently handling over 100 servers in our main monitoring and control loop, thus 42 should be quite comfortable.

Parallelize The Initialization Process

Another enhancement may be to parallelize the initialization process. Currently this takes several minutes with just 136 servers. For larger installations, the initialization process would be painfully slow. In our main control loop we are only querying a single sensor, but during the initialization process we are performing many more queries trying to determine the characteristics of the device, asking for a temperature sensor list, etc., all of which take much longer than querying a single sensor. A similar parallelization approach, as described above in parallelizing the main loop, could be taken here. That is, we can parallelize the initialization process across server racks. Each node in the rack would be initialized sequentially, but initialization would happen in parallel across multiple server racks.

Multiple Ambient Temperature Sensors

From server model to server model, ambient temperature sensors are placed in different locations. Some are close to the front bezel, others are more recessed. The downfall with the ones that are recessed, is that when under load, the reading on the ambient temperature sensor can be influenced by the heat output from the server.

Some high end servers have multiple ambient temperature sensors. A future direction may be to give the system administrator, via the web frontend, a method to select which ambient temperature sensor to monitor, or possibly aggregate the output of multiple ambient temperature sensors in an effort to get a more stable, accurate reading.

Handling Priority Jobs In Partial Shutdowns

Another future extension may be to determine the priority of jobs in the event that nodes with suspended jobs have to be shut down to further decrease heat output. Terminating the job that had been running one month over the job that had been running six months would be desirable in this situation. Similarly, terminating the job of a CS I student over the job of the research faculty professor would also be preferred. This logic would be used when a global event is detected. Currently, all jobs are suspended, and the nodes without jobs are shut down. A second step could be added - if it is determined that temperatures are still rising - that shuts down the nodes of lower priority jobs in order to give nodes with higher priority jobs a greater chance of surviving

Statistical Analysis Of Temperature Trends

As the Round Robin Databases permit us to look at historical temperature trends over time, the numbers could be analyzed for patterns that are statistically abnormal, possibly indicating a global event that could be detected before any thresholds have been exceeded. As the server room temperature can vary depending on load across the HPC environment, for this to be truly accurate, the job scheduler would likely need to be incorporated here as well. Should a large number of jobs be submitted to the cluster simultaneously, that would explain a slight rise in temperature, and if this has happened before, we should know by how much to expect the server room temperature to rise. Anything outside of what is considered statistically within allowable range could indicate an issue.

Determining If Cooling Capacity Has Been Restored

A more difficult future project may be to automatically and accurately determine when the A/C capacity has returned to 100% after a partial cooling failure, such as in the global temperature event example listed in Section 3.4. Currently it is up to the systems administrator to clear the thermal event counters in the database and bring all the nodes back online.

Extending Beyond HPC

We have also, as part of our development and testing, branched this project and are working on a version to work in a virtualization environment. Virtual hosts, like HPC nodes, generate a lot of heat, and would be another obvious implementation. Libvirt could be leveraged to manage virtual guests similar to how SGE is being used to manage jobs in this paper. We have already began to extend the Ilom class to work with other IPMI compliant devices such as the Dell DRAC.

Bibliography

[1] A Gentle Introduction with OpenIPMI. http://openipmi.sourceforge.net/IPMI.pdf. Retrieved 15 March 2012.

[2] Active Record::Base. http://api.rubyonrails.org/classes/ActiveRecord/Base.html. Retrieved 23 March 2012.

[3] Cacti Screen Shots. http://www.cacti.net/screenshots.php. Retrieved 19 April 2012.

[4] Cacti: The Complete RRDTool-based Graphing Solution. http://cacti.net/. Retrieved 15 March 2012.

[5] Daemons. http://daemons.rubyforge.org/. Retrieved 20 April 2012.

[6] Easy, proactive monitoring of processes, programs, files, directories and filesystems on Linux/Unix | Monit. http://mmonit.com/monit/. Retrieved 15 March 2012.

[7] File:Rrddemo.png. http://en.wikipedia.org/wiki/File:Rrddemo.png. Retrieved 12 April 2012.

[8] FreeIPMI - Home. http://www.gnu.org/software/freeipmi/. Retrieved 11 April 2012.

[9] Ganglia Monitoring System. http://ganglia.info/. Retrieved 15 March 2012.

[10] Graph data source items . http://www.loriotpro.com/Products/On-line_Documentation_V5/LoriotProDoc_EN/V22-RRD_Collector_RRD_Manager/V22-R4_Graph_data_source_items_EN.htm. Retrieved 12 April 2012.

[11] gridengine/gridengine - GitHub. https://github.com/gridengine/gridengine. Retrieved 23 March 2012.

[12] HVAC. http://en.wikipedia.org/wiki/HVAC. Retrieved 22 March 2012.

[13] Integrated Lights Out Manager (ILOM) 3.0. http://web.archive.org/web/20100826080254/http://wikis.sun.com/display/x64/Integrated+Lights+Out+Manager+%28+ILOM+%29+3.0. Retrieved 11 April 2012.

[14] Introduction to RRD - Round Robin Database. http://www.loriotpro.com/Products/On-line_Documentation_V5/LoriotProDoc_EN/V22-RRD_Collector_RRD_Manager/V22-A1_Introduction_RRD_EN.htm. Retrieved 11 April 2012.

[15] IPMI - Intelligent Platform Management Interface. http://www.intel.com/design/servers/ipmi/index.htm. Retrieved 11 April 2012.

[16] IPMItool. http://ipmitool.sourceforge.net/. Retrieved 11 April 2012.

[17] ISO/IEC 9075-1:2011. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681. Retrieved 20 April 2012.

[18] Junction Table. http://en.wikipedia.org/wiki/Junction_table. Retrieved 22 March 2012.

[19] Linux System Monitoring Nagios Screenshot. http://linuxscrew.s3.amazonaws.com/wp-content/uploads/2012/03/linux-system-monitoring-nagios-screenshot.jpg. Retrieved 19 April 2012.

[20] Manpage of IPMI-SENSORS. http://www.gnu.org/software/freeipmi/manpages/man8/ipmi-sensors.8.html. Retrieved 11 April 2012.

[21] Microsoft SQL Server. http://sqlite.org/. Retrieved 20 April 2012.

[22] Model EM1 - IT Environmental Monitor. http://www.sensatronics.com/products_environmental-features.html. Retrieved 19 April 2012.

[23] Monit | Screenshots. http://mmonit.com/monit/screenshots/. Retrieved 19 April 2012.

[24] MRTG Implementation Manual. http://www.enterastream.com/whitepapers/mrtg/mrtg-manual-cap6.html. Retrieved 12 April 2012.

[25] Munin - Trac. http://munin-monitoring.org/. Retrieved 15 March 2012.

[26] Munin and Nagios. http://munin-monitoring.org/wiki/HowToContactNagios. Retrieved 19 April 2012.

[27] Munin_Template_nodeview_javascript. http://munin-monitoring.org/wiki/Munin_Template_nodeview_javascript. Retrieved 19 April 2012.

[28] MySQL. http://www.mysql.com/. Retrieved 20 April 2012.

[29] Nagios - Nagios Overview. http://www.nagios.org/about/overview/. Retrieved 15 March 2012.

[30] OGF DRMAA Working Group. http://www.drmaa.org/. Retrieved 11 April 2012.

[31] OOM Killer. http://linux-mm.org/OOM_Killer. Retrieved 23 March 2012.

[32] Open Grid Scheduler. http://gridscheduler.sourceforge.net/. Retrieved 23 March 2012.

[33] OpenIPMI. http://openipmi.sourceforge.net/. Retrieved 21 March 2012.

[34] OpenPBS Public Home. http://www.mcs.anl.gov/research/projects/openpbs/. Retrieved 23 March 2012.

[35] Oracle and Sun. http://www.oracle.com/us/sun/index.htm. Retrieved 28 March 2012.

[36] Oracle Database. http://www.oracle.com/us/products/database/index.html. Retrieved 20 April 2012.

[37] Oracle Grid Engine. http://www.oracle.com/us/products/tools/oracle-grid-engine-075549.html. Retrieved 23 March 2012.

[38] Overview - Phusion Passenger. http://www.modrails.com/. Retrieved 30 March 2012.

[39] PBS GridWorks: OpenPBS. http://web.archive.org/web/20071104122012/http://www.pbsgridworks.com/PBSTemp1.3.aspx?top_nav_name=Products&item_name=OpenPBS&top_nav_str=1service.html. Retrieved 23 March 2012.

[40] PBS Professional. http://www.pbsworks.com/Product.aspx?id=1. Retrieved 23 March 2012.

[41] PostgreSQL. http://www.postgresql.org/. Retrieved 20 April 2012.

[42] Rack Unit. http://en.wikipedia.org/wiki/Rack_unit. Retrieved 27 March 2012.

[43] Relational Database Design. http://www3.ntu.edu.sg/home/ehchua/programming/sql/Relational_Database_Design.html. Retrieved 22 March 2012.

[44] RRDtool - About RRDtool. http://oss.oetiker.ch/rrdtool/. Retrieved 11 April 2012.

[45] RRDtool - rrdruby. http://oss.oetiker.ch/rrdtool/prog/rrdruby.en.html. Retrieved 15 March 2012.

[46] RRDtool - RRDtool Documentation. http://oss.oetiker.ch/rrdtool/doc/index.en.html. Retrieved 12 April 2012.

[47] Ruby on Rails. http://rubyonrails.org/. Retrieved 11 April 2012.

[48] Ruby Programming Language. http://www.ruby-lang.org/en/. Retrieved 11 April 2012.

[49] Server Room Temperature Environmental Monitoring Equipment Accessories. http://www.sensatronics.com/products-environmental-monitoring-accessories.html. Retrieved 19 April 2012.

[50] Simplified Wrapper and Interface Generator. http://www.swig.org/. Retrieved 23 March 2012.

[51] SQLite. http://sqlite.org/. Retrieved 20 April 2012.

[52] SQLite Deployment. http://sqlite.org/mostdeployed.html. Retrieved 20 April 2012.

[53] The Official YAML Web Site. http://yaml.org/. Retrieved 21 March 2012.

[54] TORQUE Resource Manager. http://www.pbsworks.com/Product.aspx?id=1. Retrieved 23 March 2012.

[55] Univa Acquires Grid Engine Expertise. http://www.univa.com/news/Univa-Acquires-Grid-Engine-Expertise. Retrieved 23 March 2012.

[56] Univa Grid Engine. http://www.univa.com/products/grid-engine. Retrieved 23 March 2012.

[57] Unix time - Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Unix_time. Retrieved 12 April 2012.

[58] webrick: Ruby Standard Library Documentation. http://www.ruby-doc.org/stdlib-1.9.2/libdoc/webrick/rdoc/index.html. Retrieved 20 April 2012.

[59] Welcome! - The Apache HTTP Server Project. http://httpd.apache.org/. Retrieved 30 March 2012.

[60] Welcome to the Son of Grid Engine Project. https://arc.liv.ac.uk/trac/SGE. Retrieved 23 March 2012.

[61] File:Relational key.png. http://en.wikipedia.org/wiki/File:Relational_key.png, feb 2004. Retrieved 22 March 2012.

[62] File:IPMI-Block-Diagram.png. http://en.wikipedia.org/wiki/File:IPMI-Block-Diagram.png, jun 2010. Retrieved 21 March 2012.

[63] Wolfgang Barth. *Nagios: system and network monitoring.* Open Source Press GmbH, 2nd edition, 2008.

[64] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language.* O'Reilly Media, 1st edition, 2008.

[65] Jeremy Lipson. jeremy04/drmaa4ruby-1.9 . GitHub. `https://github.com/jeremy04/drmaa4ruby-1.9`. Retrieved 21 March 2012.

[66] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[67] T. Oetiker. Monitoring your IT gear: the MRTG story. *IT Professional*, 3(6):44 –48, nov/dec 2001.

[68] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000.

[69] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. Pragmatic Bookshelf, 4th edition, 2011.

[70] Daniel Templeton. [GE users] Performance requesting job status using DRMAA vs. qstat? `http://arc.liv.ac.uk/pipermail/gridengine-users/2009-May/025236.html`, May 2009. Retrieved 11 April 2012.

[71] Daniel Templeton. [GE users] Is possible implementation qstat with DRMAA without job submission whit DRMAA? `http://arc.liv.ac.uk/pipermail/gridengine-users/2010-August/031829.html`, Aug 2010. Retrieved 11 April 2012.

[72] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.