An Introduction to Proactive Server Preservation in an HPC Environment

Chad Feller[1,2] Karen Schlauch[2] Frederick C Harris, Jr.[1]

[1]*Department of Computer Science and Engineering*
[2]*Center for Bioinformatics*
University of Nevada, Reno
Reno, NV.
feller@unr.edu

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 can fire off alerts for just about anything they are configured for. 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 a novel approach to server monitoring, coupled with a response system designed to deliver an intelligent response with High Performance Computing in mind.
Keywords: *System Monitoring, Environmental Monitoring, IPMI.*

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 to see if it responds, and sends requests to specified ports to ensure that services are still running on those ports. This basic approach can answer the questions of whether is 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 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 obvious, 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. Ultimately, if the OS has a kernel panic due to a hardware or software failure, the daemon is ineffective.

This paper is laid out with the remainder of this section presenting a few select examples of other monitoring systems. Section 2 goes on to discuss our novel design from an high level overview, along with an analysis of the components leveraged by our design. Section 3 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. Section 4 concludes with the current state of the project, and Section 5 discusses future work.

1.1 System and Network Monitors

Currently 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 are going to take a look at some of the existing solutions:

Nagios

According to their website, “Nagios is a powerful monitoring system that enables organizations to identify and resolve IT infrastructure problems before they affect critical business processes” [1, 2]. 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 would reside on the remote host. It also supports the ability to allow the system administrator to provide a custom response. The system administrator would have to write custom event handlers, which could 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.

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” [3]. Additionally, Monit can be used to check system resources, much like other tools.

Ganglia

Ganglia [4, 5] is a distributed system monitoring tool, designed to log and graph, both historically and in real time, CPU, memory, and network loads. It is used primarily in environments such as HPC clusters. 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.

Cacti

Cacti [6] 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.

Munin

Munin [7] 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 into Nagios, so that Nagios’ alert and response system can be leveraged.

1.2 Environmental Monitors

Environmental monitors are commonly employed in server rooms to monitor humidity and temperature, two important elements of server room climate control. While temperature is an obvious concern, humidity can also be an issue. If there is too much humidity in the air, condensation can occur. Too little humidity can enable the buildup of static electricity in the air. Traditional environmental monitors are standalone units, which may simply sound an audible alarm, or more commonly nowadays, can send out alert emails if they are network enabled. Temperature only monitors are also available, many of which are available as units meant to hang off of a server, as a USB dongle for instance, or as a network aware standalone unit capable of feeding temperature points into other system monitoring software.

2 Our 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 such as simple mail and web servers. (Although in this era of increased 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 the environmental monitor sends out an alert, is the system administrator going to be able to get to a remote console in time? Will there be enough time to login remotely, evaluate the situation, determine the best course of action, and then execute that action before the temperature crests into the critical zone causing hardware failure?

Witnessing these types of events is what motivated this project. The fact that a 1U server can be

ordered with more than 12 CPU cores makes the potential for heat creation greater than it was 4 years ago, when that same 1U server was only available with 4 CPU cores. Naturally the HVAC system is going to be upgraded to handle the additional capacity, but when the HVAC system fails, the temperature is going to rise quite a bit faster in that same server room than it did a few years ago, giving that same system administrator a lot less time to effectively respond.

Additionally, in an HPC environment, there is the consideration of jobs. Many HPC jobs run for days, weeks, or even months. Telling a researcher that they lost a month of compute time isn't easy, or pleasant. It would be challenging for the system administrator to ascertain which jobs to try to save - if possible - versus which to shut down, in a server room with rapidly escalating temperatures. Most likely the response is something along the lines of, "shutdown everything now".

But what if we can do this all more intelligently? This facilitated the thought process of creating a 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 wanted to be able to monitor not only global cooling failures, but also localized cooling failures. Those caused by perhaps a bad fan, or a piece of debris - from a server getting unboxed, for instance - that got sucked into the front of a server rack and is now obstructing airflow into one or more servers. To best detect localized failures, temperature sensors would need to be deployed on every server.

Before discussing this idea further, an overview of some key components of the environment is in order:

2.1 IPMI

Intelligent Platform Management Interface [8], 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 and manage another device. It is specifically well suited for LOM, or "Lights Out 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. As well as on some high end, specialized workstations, such as those provided by Fujitsu. They 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 off, or has locked up due to a kernel panic.

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

The IPMI system is accessible several different ways. In this work, we will be accessing it over the network. This is typically achieved one of two ways. Either 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 paper is Sun's ILOM.

2.2 BMC

Some vendors will refer to their IPMI system as a BMC. Technically the BMC, or Baseboard Management Controller, is the core of the IPMI subsystem, it is the microcontroller that ties the whole IPMI system together. See Figure 1.

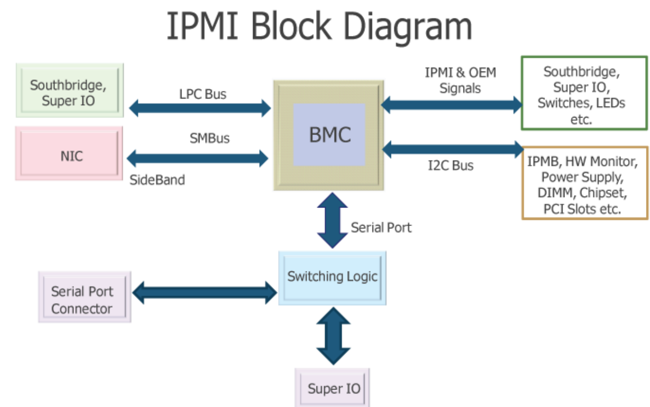

Figure 1: IPMI Block Diagram (courtesy of [9]), illustrating the Baseboard Management Controller as the center of an IPMI system.

2.3 Schedulers

Job schedulers are a key component of HPC environments. They ensure fair and equal job scheduling - that no single job starves, and that no single job consumes all system resources.

Two prominent schedulers are SGE (Sun Grid Engine) and PBS (Portable Batch System).

A batch job scheduler can be configured several different ways, depending on the preferences of the system administrator, or the requirements handed to the system administrator by researchers or management. In a most basic configuration, each user can submit jobs into an 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 more priority over others, allowing low priority jobs to be temporarily suspended while high 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.

Servers in an HPC environment, are commonly referred to as "compute nodes", or just "nodes". In this paper, the term "node" and "server" can be thought of interchangeably.

The job scheduler in use for the environment in this paper is Sun Grid Engine version 6.1u4.

2.4 Putting it together

Our idea revolves around the idea of tying the IPMI systems into a larger system, giving us their included temperature monitoring 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 IMPI 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.

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.

3 Implementation

The implementation of this project occurred in stages. First we had to find the best way to talk to the ILOMs. OpenIPMI has Python 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, so other options were explored.

We knew that we could get exactly the information we wanted from IPMItool, 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 was a desirable choice on a number of levels, so the decision was made to use Ruby 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 databases are exceptionally well suited for time series data, which in this case, is the temperature values. 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.

3.1 Ilom class

The ILOM Ruby class object grabs a bunch of information upon being initialized, as illustrated by Figure 2.

```ruby
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 2: 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 the need for them every time. If the instance variable is set, we just send that workaround along with the FreeIPMI command.

@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 things are good 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 part 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, it 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.

3.2 IlomRRD class

This class sits on top of the official RRDtool Ruby bindings [10]. This class serves two main purposes.

First, the class creates and updates RRD databases every minute when new temperature values are polled. These values are written to an RRD database for that particular server. Each server has its own RRD database.

Secondly, the class knows how to create graphs from the RRD databases. 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.

Values required for RRDtool create and update the RRD databases, as well as graph the temperature values are read from a YAML [11] configuration file. Values such as RRD 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.

3.3 IlomSGE class

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

An issue that prevented us from 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.

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.

3.4 Database backend

The backend consists of an RRD databases for storing time series data (temperature values). As well as SQL databases for recording server characteristics server location. The Ruby ActiveRecord gem is used to leverage the SQL database. There are two main Ruby models that we are interacting with. A Node model, and a Location model.

The Node model largely contains the same information as the Ilom class, discussed in Section 3.1, but here is stored persistently. It also contains additional values for dealing with thermal events.

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. A foreign key ID is what allows a Location to find its corresponding Node.

3.5 Rails frontend

The system frontend is illustrated by Figure 3.

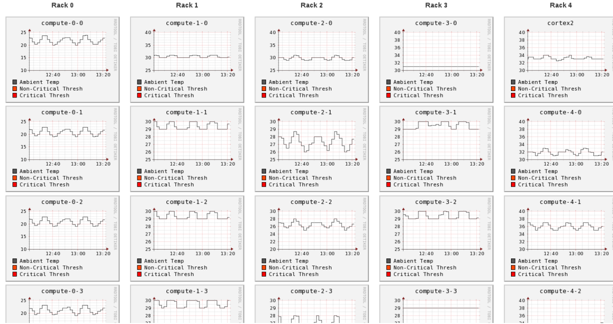

Figure 3: The default frontend view, RRDtool generated temperature graphs of all of the nodes. The layout of the graphs on the webpage match the physical layout of servers in the server room. Clicking on any of the graphs will bring up a larger graph, along with other characteristics of the server, pulled 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.

3.6 Runtime

A runtime data structure is assembled, internally called a *nodelist.* The each element in the nodelist is a nested has structure as shown in Figure 4.

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

Figure 4: The structure of a *nodelist* entry

So each entry in the *nodelist* can reference its Ilom object information, its IlomRRD object information, and its NodeID for database access.

The *nodelist* is used by the main loop of the program. The main loop of the program is where the backend daemon lives after initialization, until termination.

A high level overview of the logic of the backend daemon is as follows:

- During each loop, which runs once every minute, every Node is addressed. Upon completion of addressing every Node, the loop sleeps until the remainder of one minute has elapsed.

- At the beginning of the loop, an inner loop is run, where each Node in *nodelist,* is polled via IPMI, and temperature values are updated in SQL and RRD databases accordingly. If it is a 5th loop, RRD graphs are updated. If a thermal threshold has been reached - that is if the ambient temperature has exceeded one or more thresholds - appropriate actions are taken. Note that any actions taken here would be local thermal event actions.

- After each entry in *nodelist* has been addressed, determine if there is a global temperature event taking place, and if so take appropriate actions to deal with a thermal event.

4 Conclusion

This project was tested throughout development process. Each component has been tested quite a bit in a standalone capacity before it was integrated with the rest of the code.

Since 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 we added on 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 IMPI devices can and will return garbage values from time to time. We learn to handle these types of situations gracefully, so that we don't crash the system.

After months of work, much trial and error, many code revisions, and database model changes, everything has finally come together.

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.

More details and illustrations on this work can be found in [12].

5 Future Work

We have identified numerous future directions for this project. A lot of the future direction would just be the next logical iteration for this work.

From server model to server model, ambient temperature sensors are placed in different locations. Some 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 work would 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.

Another future extension would be to determine the priority of jobs in the event that nodes with suspended jobs had to be shut down to further cut heat output. Killing the job that had been running 1 month over the job that had been running 6 months would be desirable in this situation. Similarly, killing the job of a CS I student over the job of the research faculty professor would also be preferred.

A more difficult future project would be to able to automatically and accurately determine when the A/C capacity has returned to 100% in the event of a partial cooling failure, such as in the global temperature event example listed in Section 2.4.

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.

Acknowledgements

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

References

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

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

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

[4] Ganglia Monitoring System, Retrieved 15 March 2012, from http://ganglia.info/

[5] Matthew L Massie, Brent N Chun, David E Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Computing, Volume 30, Issue 7, July 2004, Pages 817-840, ISSN 0167-8191, 10.1016/j.parco.2004.04.001.

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

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

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

[9] *Wikipedia* (16 June 2010). Retrieved 21 March 2012, from http://en.wikipedia.org/wiki/File:IPMI-Block-Diagram.png

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

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

[12] Chad Feller, "Beyond Monitoring: Proactive Server Preservation in an HPC Environment", Masters thesis, University of Nevada, Reno, May 2012.