Monitoring with MonAMI: a case study.

A Paul Millar[1], G A Stewart[1], G A Cowan[2]

[1] Dept. of Physics and Astronomy, University of Glasgow, University Avenue, Glasgow, G12 8QQ, UK

[2] Department of Physics, University of Edinburgh, Edinburgh, UK

E-mail: p.millar@physics.gla.ac.uk

Abstract. Computing resources in HEP are increasingly delivered utilising grid technologies, which presents new challenges in terms of monitoring. Monitoring involves the flow of information between different communities: the various resource-providers and the different user communities. The challenge is providing information so everyone can find what they need: from the local site administrators, regional operational centres through to end-users.

To meet this challenge, MonAMI was developed. MonAMI aims to be a universal sensor framework with a plugin architecture. This plugin structure allows flexibility in what is monitored and how the gathered information is reported. MonAMI supports gathering statistics from many standard daemons, such as MySQL and Apache, and system data, such as network sockets and memory consumption. The gathered data can be sent to many monitoring systems, including Ganglia, Nagios, MonALISA and R-GMA.

This flexibility allows MonAMI to be integrated into whatever monitoring system is being used. This avoids the current duplication of sensors and allows the gathered statistics to be presented within a greater diversity of monitoring systems.

Using the MonAMI framework, sensors have been developed for the DPM and dCache storage systems, both common at HEP grid centres. The development of these tools specifically to tackle the challenges of high availability storage is described. We illustrate how MonAMI's architecture allows a single sensor to both deliver long term trend information and to trigger alarms in abnormal conditions.

The use of MonAMI within the ScotGrid distributed Tier-2 is examined as a case study, illustrating both the ease with which MonAMI integrates into existing systems and helps provide a framework for extending monitoring where necessary.

1. Introduction

Grid computing has now emerged as a viable system for making vast amounts of resources, both CPU and storage, available to a geographically distributed community of users. This is achieved by aggregating the available resources of many resource-provider sites within a system that hides which particular resource-provider is in use.

Within most grids sites exist independently of each other. Although currently available software stacks generally require a certain conformity of platform the sites remain largely autonomous, interacting with the higher levels of the grid through well defined services. The decision of what fraction of available resources any subset of users will be allowed is delegated to the site. In an idealised grid environment, no site is more important, apart from at a trivial level (e.g., amount of resource provided or provision for usergroup-specific requirements).

Like any computational resource, Grid resources can suffer from performance problems or even complete outages. Sometimes end-users can precipitate problems by following unexpected

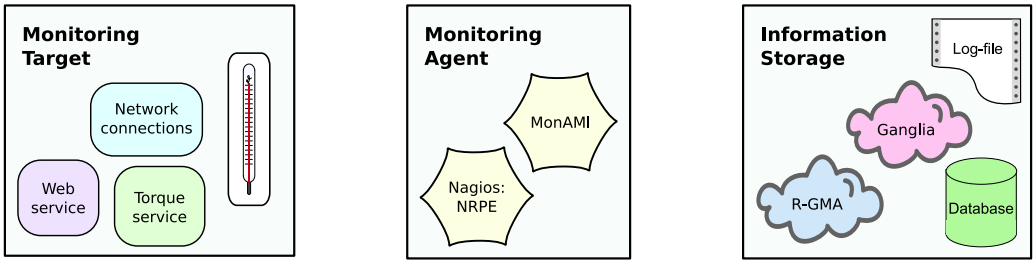

Figure 1. The three-component model for basic monitoring. The three solid boxes represent the three components; information flows from left to right. The boxes contain specific examples of each type.

usage patterns. These unusual usages can cause greatly reduced performance or may even cause services to fail. Under these circumstances it is important to establish what triggered a problem so it can be prevented in the future.

None of these requirements are unusual in computational services. What makes monitoring in a grid environment challenging is the interaction of the different geographically distributed groups of people.

1.1. Monitoring in a Grid environment

Monitoring is the flow of information from one community to another. The most obvious flow of information is from the various resource-providers (the sites) and the different user communities.

However, there are other flows of monitoring information. Resource-providers may be grouped into regional or national groups. Users, grouped into Virtual Organisation (VOs), might provide monitoring information, describing how well they perceive the grid working. Automated tests might conduct a systematic study of the current Grid status, feeding this information both to the individual site administrators and to VOs.

In general, grid monitoring can be characterised in terms of collecting data from multiple geographically distributed information sources and providing it to multiple geographically distributed interested parties.

Any one actor may be a member of multiple groups of interested parties. Each group may require a different subset of all available information, or may require the same low level information to be aggregated in a different way. The challenge is providing information so everyone can find what they need: from the local site administrators, regional operational centres through to end-users. This is what MonAMI aims to support.

2. Understanding monitoring

Before discussing how to support monitoring to many different groups, it is beneficial to discuss what is meant by "monitoring". At the most basic level, monitoring is watching something; that is, maintaining some knowledge about the state of a particular service. This knowledge is achieved by the flow of information from whatever is being monitored into some storage of information. This flow of information will be discussed.

We start from a simple model of monitoring interaction: the three-component model. As the name suggests, in this model there are three components with basic monitoring. These are the monitoring target, the monitoring agent and the information storage. These are shown diagrammatically in Figure 1 along with examples of each type.

The monitoring target is the system we wish to maintain knowledge. This could be some hardware component (a fan, a network switch, an air-conditioning unit) or some software service (for example, whether the software is working correctly or some performance metrics).

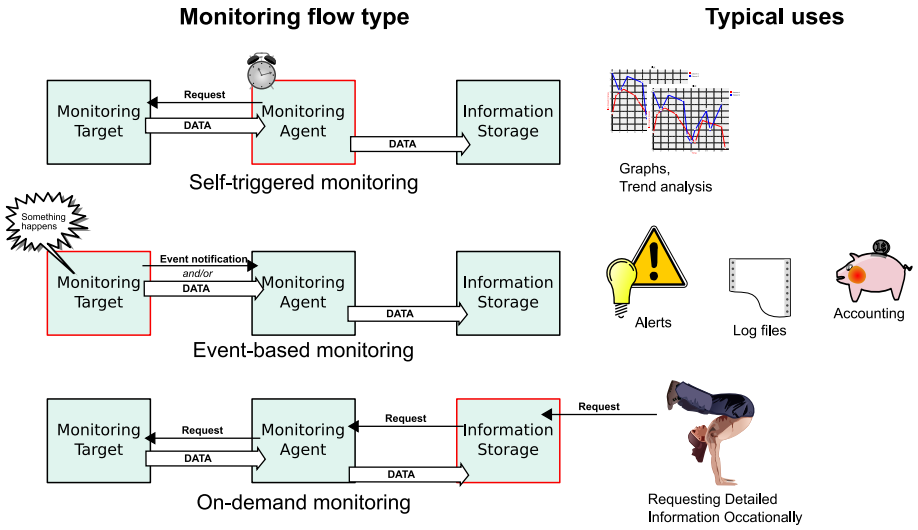

Figure 2. The three basic monitoring flows: self-triggered, event-based and on-demand.

The second component is the monitoring agent. This is the software that achieves the flow of information from the monitoring target into the information storage. Whatever the monitoring target, there must be some means by which the monitoring agent can interact with it. This will be monitoring-target-specific, so the monitoring agent must know how to extract information from the monitoring target.

The final component is the Information Storage. An information storage system will typically have some (usually non-volatile) storage with some means by which information can be added. Examples of information storage systems include log-files, databases (such as MySQL[1]), site-local monitoring systems (e.g., Ganglia[2] and Nagios[3]) and distributed information systems (such as MonALISA[4] and R-GMA[5]).

It should be emphasised that many systems provide their own monitoring agent or infrastructure for remote monitoring; for example, Nagios provides NRPE[6] and NSCA[7], Ganglia provides gmetric[8] and MonALISA provides ApMon[9]. In this respect, MonAMI can be seen as unifying monitoring as a service and, under this unified MonAMI-centric view, these other systems are Information Storage components.

2.1. Basic monitoring flows

Using this simple three-component model, there are three natural monitoring flows.

Traditional analogue monitoring systems work by exploiting some physical change caused by the quantity one wants to monitor; for example, the temperature of a room may be monitored by observing the thermal expansion of a liquid. Such physical monitoring systems results in a continuous monitoring process: ignoring response times, a room's temperature is knowable at arbitrary short time intervals.

Computer activity, and by extension the monitoring information this activity provides, will occur at specific times. In contrast to physical monitoring, computer-based monitoring is triggered. The trigger is either by an internal clock or else by some external factor.

The three-component model provides a natural categorisation of monitoring activity, leading to three monitoring flows. The distinction is which of the three components triggers the monitoring activity. The three monitoring flows are shown diagrammatically in Figure 2.

Self-triggered monitoring is where the agent triggers the process of gathering information independently of external influences. This is style of monitoring is most often used to obtain a

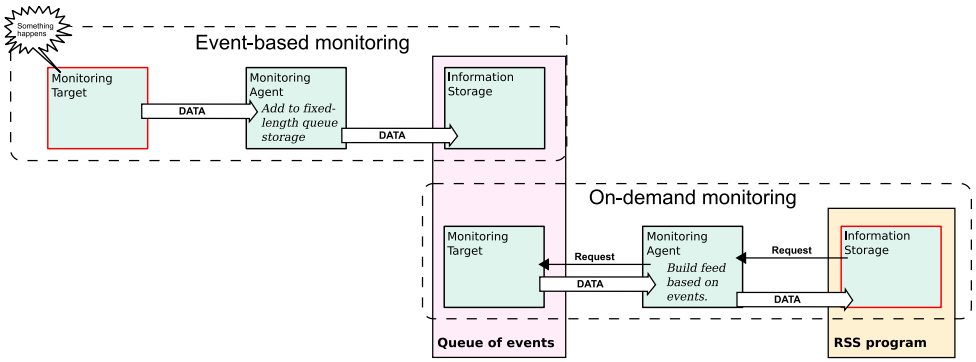

Figure 3. An example of compound monitoring flows. The two different monitoring flows involved with an RSS alert system are shown: event-driven and on-demand.

regular set of measurements, every t_{poll} seconds. If the gathered data is examined with timescale $t \gg t_{\text{poll}}$ then many of the artifacts from monitoring in discrete time will be reduced and the classic analogue monitoring is emulated.

Another monitoring flow is where the monitoring target triggers the monitoring activity. This is most often used for state-transition or event-based monitoring; for example, to capture information about a file transfer.

The final monitoring flow, on-demand monitoring, is triggered by some external agent. Without loss of generality, we can include this agent within the information storage component so consider this flow as being triggered by the storage component. On-demand monitoring allows the information system to requests a subset of the available data at a times under the information system's control. KSysGuard[10], part of KDE[11], illustrates this monitoring flow.

2.2. Compound monitoring

The three-component model is sufficient to describe some simple monitoring interactions. However, more complex monitoring can be described by aggregating multiple 3-component models, considered as a pair-wise list of monitoring components with increasing dependency.

Each pair of three-component models are combined by considering the information system of the first three-component model as the monitoring target of the less fundamental three-component model. Fundamental here is merely an ordering within the list of three-component models, but typically the more fundamental three-component model will operate correctly without the less fundamental model.

The following gives an illustration of combining two three-component models. Consider a system that logs alarm events that maintains a fixed-length queue of events from which the XML of an RSS-feed is built dynamically. This would allow people to subscribe an RSS aggregator to receive notification of these events. The complete system can be understood as two three-component models with different monitoring flows and is shown in Figure 3.

The first three-component model using the event-driven monitoring flow. The agent receives the alarm events and storing the information within the fixed length queue. The second three-component model uses on-demand monitoring flow to build suitable XML when a user's RSS aggregator refreshes.

3. MonAMI

This section introduces the MonAMI monitoring agent. It starts by discussing the motivation for writing the agent, followed by a succinct overview of its architecture. Datatrees are introduced in §3.2 and then the three data flows are mapped to MonAMI concepts in the following section.

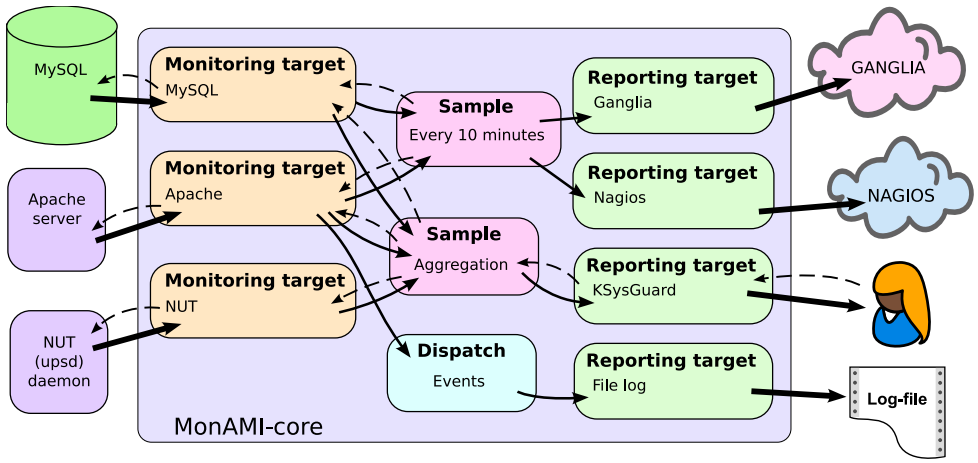

Figure 4. Shows MonAMI example configuration, illustrating the architecture. Small arrows show requests for information. Thicker arrows show the flow of information.

This section completes with a list of the monitoring plugins mentioned in §4 along with a short description of each, allowing the reader to better understand the case-study.

There are many existing monitoring agents and frameworks. A recent report[12] included a review of existing monitoring systems. It showed that they all suffer from limitations that prevent the necessary interoperability for Grid-wide monitoring. For example, some provide only limited information (e.g., that a service is in a `OK`-, `Warning`- or `Critical`-state). Most agents provide information only to one information system.

Perhaps the most challenging limitation is that the existing monitoring agents support only one (or, at most, two) of the three monitoring flows. The three flows are architecturally different. Adding supporting for additional monitoring flows retrospectively is difficult and would require substantive rewriting of the existing code-base.

MonAMI was written to overcome the lack of comprehensive monitoring. It provides a monitoring agent that supports the three monitoring flows. It uses a plugin structure both for gathering information and for making that information available. This allows it to be extended easily. Figure 4 illustrates this, showing the interaction between different plugins for a configuration involving the three monitoring flows.

In this sense, MonAMI is a universal monitoring agent: the three monitoring flows are supported and any missing functionality can be added. A set of useful plugins are provided and new plugins can be added to support gathering information from a different system, or reporting information to additional information systems.

3.1. Structure

The MonAMI agent has two main parts: MonAMI-core and a set of plugins. MonAMI-core is a multi-threaded, asynchronous, message-passing framework that instantiates the various plugins, provides some common functionality and mediates communication between plugin-instances. It is also responsible for shutting down in a controlled fashion so no data is lost.

Plugins have a very simple API that allows interaction whilst hiding the complexity of MonAMI-core. Plugins come in two types: either singleton or configurable.

Singleton plugins will have exactly one instance and provide some stateful service for the other plugins. Examples include the `timekeeper` plugin (providing a subscription service for timed events) and the `logwatcher` plugin (a subscription service for receiving file updates).

Configurable plugins are instantiated by MonAMI-core only if they are mentioned within a

configuration file. Distinct from the singleton plugins, configurable plugins can be instantiated multiple times. Each instance must have a unique name and each has a set of attributes (keyword-value pairs) that configuration the instance's operation. All of MonAMI's external interactions are mediated through configurable plugins.

Configurable plugins can be further subdivided into three groups: those that monitor something (e.g., `torque` and `dpm`), those that store information somewhere (e.g., `ganglia` and `rgma`) and those that mediate the flow of information (e.g., `sample` and `dispatch`).

3.2. Datatrees

A datatree is a collection of related metrics arranged into a tree structure. Related is a loose term, but usually means that the metrics were collected at the same time. A monitoring plugin, when requested, might provide a datatree representing the current status of some service.

The metrics are collected into a tree structure. Different branches represent different aspects of the service being monitored. This is to facilitate selection of sub-trees. An example datatree is illustrated in Figure 5.

Monitoring information is passed between plugin instances as datatrees. The plugin instances might process the data (modify the data, aggregate different sources, etc) or store the information somewhere.

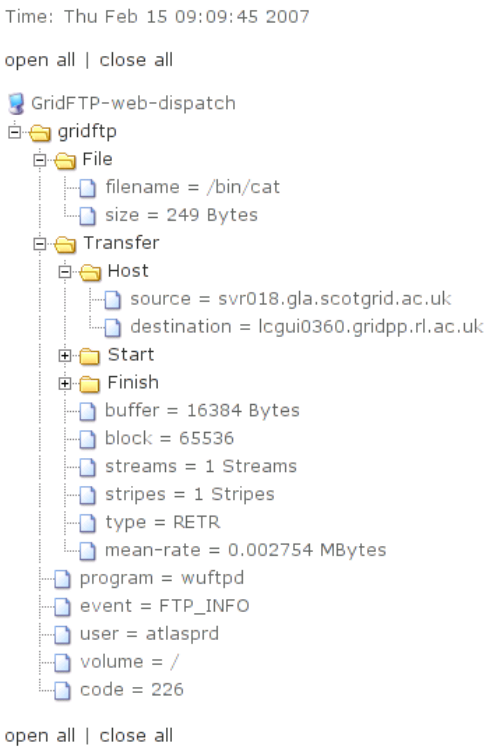

Figure 5. An example datatree showing some of the available metrics from a Grid-FTP transfer.

3.3. Data flows

For self-triggered monitoring flows (e.g., periodic monitoring), a `sample` plugin instance will be triggered by the `timekeeper` plugin. Triggering events using the `timekeeper` allows for an orderly shutdown: stop the timekeeper then wait for all activity to stop.

When triggered, the `sample` plugin will request data from multiple sources, select requested sub-trees and aggregate the resulting set of datatrees into a larger datatree. The resulting datatree is sent to one or more reporting plugin instances.

Event-base monitoring flow is similar: a datatree is sent from a monitoring plugin instance to those plugins that have subscribed to events. These are then sent on to the reporting plugin instances and the information is stored.

Note that the reporting plugins can be used for either self-triggered or event-based monitoring flows without requiring any changes. One can even configure the same reporting plugin instance to accept both self-triggered and event-based monitoring information concurrently.

The on-demand monitoring flow requires a reporting plugin that sends requests for data to the configured data sources. This can be a monitoring plugin or `sample` plugins. Again, plugins are reused. A monitoring plugin can be used for self-triggered and on-demand monitoring flows without any change, and both monitoring flows can occur concurrently with MonAMI-core maintaining concurrency and data-caching policies.

3.4. Description of plugins
The MonAMI codebase includes many useful plugins. Other plugins can be added simply by installing the plugin files within the correct directory. This section briefly describes the monitoring plugins mentioned within the case-study: a subset of the available plugins.

3.4.1. DPM The Disk Pool Manager (DPM) project [13], [14] is based at CERN. It provides software that makes disk storage available via WAN/Grid protocols and local LAN protocols as part of the EGEE project [15]. There is currently limited status monitoring available through the DPM API. To monitor the DPM service, the plugin must query the database DPM uses.

DPM running on disk-pool nodes provides a log-file of WAN-level transfers by maintaining a log-file of Grid-FTP transfers. The MonAMI gridftp plugin understands this format and uses the log-file to provides event-based monitoring. Each event is a datatree for a Grid-FTP transfer, containing the information about the transfer.

3.4.2. dCache The dCache project[16] provides a storage solution. It is a joint project involving teams from Fermilab and DESY. The dCache service provides access to both disk and tape storage and understands several WAN and LAN protocols. A dCache monitoring plugin for MonAMI was developed in order to monitor the SRM transactions. Due to the lack of monitoring API, this was only possible by querying the database directly.

3.4.3. GridFTP GridFTP[17] extends FTP to support data movement operations within grid infrastructure. A reference implementation is available as part of the Globus Toolkit[18]. A monitoring plugin for GridFTP has been developed that provides event-based monitoring of transfers. This is used to provide information for the Grid-wide monitoring systems GridVIEW[19].

3.4.4. TCP The `tcp` monitoring plugin provides a flexible way of monitoring network connections. It counts the number of network connections that match some search criteria. For example, one can count the number of incoming connections to a specific port, the number of connections to any port from any host that are in a particular state, or the number of connections from a specific host.

3.4.5. Torque The Torque project[20] is the open-source fork of the Portable Batch System (PBS) software. The software provides a batch management framework where jobs are submitted and can be scheduled to run on worker-node computers. The `torque` monitoring plugin queries the torque server to discover the current status of the jobs, queues and overall torque server.

3.4.6. Maui Maui[21] is a popular system that provides advanced scheduling algorithms[22][23] for batch systems, often used in conjunction with torque to control activity on a computer cluster. It includes a concept of "usage" and target "fair-share" at different levels of granularity (user, group, class, etc.)

3.4.7. MySQL MySQL[1] is a popular, widely used database system. It provides monitoring facilities by extending the SQL language with monitoring-specific queries. The MonAMI plugin uses these extensions to build datatrees that provide a comprehensive description of the current status of the database.

4. ScotGrid case study

ScotGrid [24] is a distributed Tier-2 site within EGEE. It has component sites located at Glasgow (`UKI-SCOTGRID-GLASGOW`), Edinburgh (`ScotGrid-Edinburgh`) and Durham (`UKI-SCOTGRID-DURHAM`). Each site is different and has unique aspects that they bring to the Tier-2. A summary of the experiences of these sites is given below.

4.1. Glasgow

The site at Glasgow underwent a considerable expansion in 2006, from a 26 node batch system with 2 disk servers, to one with 140 batch workers and 10 disk servers hosting 100TB. This increase in the scale of the site brought new challenges to the system administration staff, as previous *ad hoc* methods of running the cluster were no longer appropriate.

One of the first challenges faced was how to monitor the site's new, much larger, storage system. This was especially problematic as the recently developed Disk Pool Manager (§3.4.1) had no well developed monitoring API or dedicated monitoring applications. Simple monitoring based on the occupation of disk systems by files gives no information as to how owns the file, as all files on disk are owned, in the Unix sense, but the DPM system itself. In this case the site's requirements to monitor storage usage on a per-VO basis was fed directly to the MonAMI developers, who then worked closely with the site to ensure the deployment was correct. Once the monitoring system had been deployed it was far easier for the site to monitor the ingest of files to the site, with MonAMI reporting to Ganglia giving a simple RRD based historical view. As the result of this successful deployment the GridPP[25] Storage Group recommended MonAMI monitoring for Tier-2 centres using DPM.

MonAMI was further used to help the site to recognise, and take remedial action, when problems arose with the R-GMA software. In this case an excessive number of network connections to R-GMA was observed, causing the R-GMA site installation to lock up, which then prevented the site from publishing accounting data to the grid. Previously this situation was usually only corrected by EGEE CIC-On-Duty team members raising a ticket against the site. However, by utilising MonAMI's ability to monitor the number of active TCP connections abnormal numbers of R-GMA connections could be identified in an automatic way. This was very simple for the site to configure. Then reporting to Ganglia, for an RRD historical view, and to Nagios, to provide an alarm system, ensure that the site never falls behind in the publishing of accounting data. This has improved the reliability of the site, and prevented the need for escalation through high layers of grid operations.

With the Glasgow site now running a much larger system scaling problems in torque and the compute element gateway to the site started to occur. In one incident the submission of several 1000 jobs through the lcg-CE caused the batch system (§3.4.5) to lock-up, denying access to resources to all users and causing jobs to fail in their epilogue, resulting in wasted cycles. The graphs captured at this time are shown in Figure 6. The visible gap is due to the excessive time Torque and Maui took to provide monitoring data. When sending metric update messages, MonAMI ganglia plugin provides an updated estimate of after what time Ganglia can drop a metric as no further monitoring data will be provided. With a server suffering rapidly increasing load, Ganglia may drop metrics before new messages (with the adjusted metric life-time estimates) are received. However, after the first such message there are no further gaps. This is whilst the server was suffering a 1-minute load average of over 300 and ssh daemon was

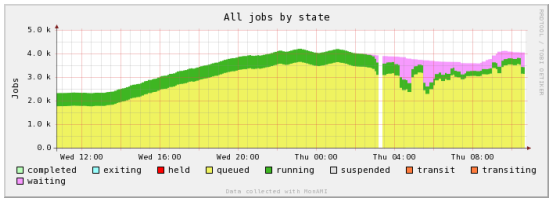

Figure 6. Ganglia graph showing MonAMI-collected data. The effects of too many jobs is shown.

unable to fork before the ssh client timed out, preventing site administrators from logging into the server remotely.

Close contact with the developers of MonAMI resulted in capturing accurately the site's requirement for batch system monitoring. In particular a break down of job information on a per-queue basis was very desirable. Further, monitoring job states was essential as an early indicator of problems was significant number of jobs entering the Torque `Waiting` state. On a number of occasions this detailed monitoring has allowed site administrators to take preemptive action to delete problematic jobs, or ban problematic users, ensuring continued service for other users.

Recently the development of iGoogle[26] Gadgets[27] of the monitored Torque information in Ganglia has allowed staff to integrate batch system monitoring into their "homepage" view on Google. While it may seem trivial accessing site monitoring information in such a novel way really does help to aid the system administration team in providing a reliable service.

4.2. Edinburgh

Edinburgh currently uses dCache as their production storage element. MonAMI was used to monitor the SRM transactions until the recent schema change.

However, MonAMI continued to be used to monitor the status of the dCache server through the use of the non-dCache specific plugins. For example, since dCache is a storage element, new TCP connections are continually being opened in order to transfer data to and from its GridFTP servers. The MonAMI tcp plugin proved invaluable when coping with a bug within the dCache GridFTP server. This bug led to some TCP connections associated with a transfer remaining in a half-closed state where the remote end closes their connection but the local end remains open (the `CLOSE_WAIT` state). A build up of `CLOSE_WAIT` sockets eventually led to resources exhaustion on the server and a corresponding loss of service.

Using MonAMI, the status of these connections could be published into the Edinburgh Ganglia system, allowing the build up of `CLOSE_WAIT` sockets to be easily tracked and correlations to be discovered. This accumulation of evidence proved essential to the dCache developers in identifying the problem and allowed them to develop a solution.

Whilst the dCache `CLOSE_WAIT` issue was still present, MonAMI could help to provide a reliable service. Using the nagios plugin, MonAMI can alert when the number of half-closed sockets is reaching a dangerous level. This allows preemptive intervention prior to the number of `CLOSE_WAIT` sockets adversely affecting the service.

When the new computing and storage hardware is deployed at Edinburgh, we intend on deploying MonAMI to monitor both our new DPM storage element and the batch system. We are confident that this will lead improved site availability.

4.3. Durham

Following the development of MonAMI plugins useful to grid sites, and especially following the deployments at Glasgow, Durham realised that significant value could be obtained from their own deployment of MonAMI. Here MonAMI has been used to monitor their DPM and batch system, helping to provide useful information to site managers as well as to the system administration team.

In addition, Durham had a somewhat novel grid batch system, where local jobs were able to suspend grid jobs. It was easy to add support for monitoring suspended jobs in the MonAMI plugins, so now site staff can see this information at a glance.

5. Conclusions

At the heart of MonAMI is a flexible monitoring agent that powers a generic monitoring framework. The agent provides a service where monitoring information is made available to

multiple monitoring systems using an extensible set of plugins. This has been demonstrated with MonAMI providing monitoring data to Ganglia, Nagios, KSysguard and R-GMA.

A key feature of MonAMI is usability, allowing site administrators to integrate MonAMI-provided monitoring information into existing monitoring infrastructures. All three of the ScotGrid sites have adopted MonAMI as a core part of their monitoring infrastructure, customising it to their particular needs.

The MonAMI agent has been shown to degrade gracefully under extreme load conditions. It continued to provide monitoring information while enforcing concurrency and data-caching policies; this was achieved while other daemons provided only intermittent service.

Future work will include providing support for plugins written in Python. This is to reduce the barrier for people contributing plugins, allowing site-administrators to write custom plugins easily.

Acknowledgments

Funding for this work was provided by STFC/PPARC via the GridPP Project. One of the authors (G A Stewart) is funded by the EGEE project.

References

[1] MySQL *http://www.mysql.com/*
[2] Massie M L, Chun B N and Culler D E 2004 The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing* **30** 817–840
[3] Nagios *http://www.nagios.org/*
[4] Newman H B, Legrand I C, Galvez P, Voicu R and Cirstoiu C March 2003 MonALISA: a distributed monitoring service architecture *Comp. in High Energy and Nuclear Phys. (CHEP03) http://monalisa.caltech.edu/documentation/MOET001.pdf*
[5] Cooke A *et al* 2003 R-GMA: an information integration system for grid monitoring *LNCS* **2888** 462–481
[6] Nagios NRPE documentation *http://nagios.sourceforge.net/docs/nrpe/NRPE.pdf*
[7] NSCA (with Nagios) *http://nagios.sourceforge.net/download/contrib/documentation/misc/NSCA_Setup.pdf*
[8] GMetric repository *http://www.ganglia.info/gmetric/*
[9] ApMon: Application Monitoring API *http://monalisa.cacr.caltech.edu/monalisa__Service_Applications__ApMon.html*
[10] Schlaeger C 2000 The KSysguard Handbook *http://docs.kde.org/stable/en/kdebase-workspace/ksysguard/*
[11] KDE: K Desktop Environment *http://kde.org/*
[12] Albrand S, Duellman D and Millar A P Metadata monitoring requirements *http://www.gridpp.ac.uk/datamanagement/metadata/Documents/MonitoringReq.pdf*
[13] Disk Pool Manager. *http://twiki.cern.ch/twiki/bin/view/LCG/DpmAdminGuide*
[14] Abadie L, Frohner A, Baud J-P, Smith D, Lemaitre S, Nienartowicz K and Mollon R. Sept. 2007 DPM status and next steps *Comp. in High Energy and Nuclear Phys. (CHEP07) http://indico.cern.ch/contributionDisplay.py?contribId=389&sessionId=21&confId=3580*
[15] Enabling Grids for E-sciencE. *http://www.eu-egee.org/*
[16] dCache Mass Storage System. *http://www.dcache.org/*
[17] GFD-20 Allcock W 2003 GridFTP: Protocol Extensions to FTP for the Grid *http://www.ogf.org/documents/GFD.20.pdf*
[18] The Globus Toolkit (GT) *http://www.globus.org/toolkit/*
[19] GridVIEW *http://gridview.cern.ch/*
[20] Torque resource manager *http://www.clusterresources.com/pages/products/torque-resource-manager.php*
[21] Maui cluster scheduler *http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php*
[22] Bode B *et al* 2000 The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters. 4^{th} *Annual Linux Showcase & Conference, http://www.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/bode/bode_html/*
[23] Jackson D, Snell Q and Clement M. 2001 Core algorithms of the Maui scheduler, "Job Scheduling Strategies for Parallel Processing", *LNCS* **2221** 87–102
[24] The ScotGrid Project. *http://www.scotgrid.ac.uk/*
[25] The GridPP Project. *http://www.gridpp.ac.uk/*
[26] iGoogle: interactive Google *http://www.google.com/ig/*
[27] Google Desktop Sidebar Scripting API *http://desktop.google.com/script.html*