The Node Monitoring Component of a Scalable Systems Software Environment

Sam Miller and Brett Bode
Department of Electrical and
Computer Engineering
Iowa State University
Scalable Computing Laboratory
Ames Laboratory, U.S. DOE
Ames, Iowa 50011
Email: {samm, brett}@scl.ameslab.gov

Abstract

We describe Fountain, an implementation of the Scalable Systems Software node monitor specification targeted at aggregate node monitoring for clusters. Fountain is designed from the ground up as a hierarchical system with scalability in mind. It leverages widely used technologies such as XML and HTTP to present an interface to other components in the SSS environment. In this paper we describe the design choices of Fountain and discuss some preliminary performance measurements on medium sized clusters.

1 Introduction

The Scalable Systems Software (SSS) center [7] is a multi-institutional initiative to design and build component based cluster management software to more effectively utilize terascale computational resources. The goal of the center is to develop open source components that work effectively on small and large scale systems. As computational resources grow beyond today's teraop class systems into future petaop and beyond systems, scalability problems must be solved to utilize these resources to their full potential. Fault-tolerance, reliability, manageability, and ease of use for both system administrators and users is a key goal of the SSS project. Presently, the SSS project has defined interfaces and created reference implementations for a variety of components. The interface for each component is publicly documented since it is unlikely that every component will fit the needs for every installation site.

In this paper we present the design of a node monitoring component called Fountain. Section 2 presents some background and motivation for designing this component. Section 3 gives an overview of existing node monitoring tools. Section 4 describes the interface to a node monitor and what services Fountain provides. Section 5 discusses the design choices and implementation of Fountain. Section 6 gives results for the test environments. Section 7 proposes some future work to add new features to Fountain. Lastly, section 8 concludes this paper.

2 Background and Motivation

System level monitoring is an important part of cluster management. As computational resources scale into the thousands of nodes per system domain, the failure rate for individual components will increase. Detecting these component failures before they happen requires an accurate snapshot of the cluster's status so the system administrator and other system components can act appropriately. Presently, node monitoring for a wide number of clusters is extremely primitive. The most often used method is the ping command, which does little more than tell the administrator if a node is physically connected and powered on. It still may be unusable, or unreachable due to a number of problems. In the event the ping command fails, the typical solution is to reboot the node, potentially destroying any temporary logs or event traces that preceded the failure. This methodology is inefficient for even medium sized clusters. As clusters continue to increase in size, effective monitoring of system and node information will be required.

Fountain has three distinct design goals in order to be a reliable, accurate, and effective node monitor. First, it should be fault tolerant in the sense that it should handle both individual and multiple node failures. In this context a node failure is defined as a loss of power, kernel panic, or a similar loss event. The second design goal of Fountain is that it should have a low memory footprint and processing requirements for each node. Monitoring perturbation is an unavoidable side affect of any performance monitoring system, however Fountain strives to reduce this as much as possible. The third and final design goal is Fountain should be able to effectively scale to next generation computational resources containing thousands of nodes.

The motivation to design Fountain came from the need to provide the cluster scheduler of the SSS environment with an accurate snapshot of each node's status. The cluster scheduler requires this information so user jobs can be scheduled and run accordingly. This is why fault tolerance is the primary and most important design goal of Fountain. In order to reliably provide the scheduler with accurate information, Fountain needs to be able to both recognize and recover from individual and multiple node failures in an efficient, reliable, and effective manor without adversely affecting the system.

3 Prior Work

Many monolithic resource management systems include basic monitoring capabilities. Products such as the Portable Batch Scheduler, LoadLeveler, Platform LSF, and the Sun Grid Engine all provide monitoring interfaces for cluster administration. While these systems all provide adequate monitoring features, many are not capable of interfacing with the existing SSS components shown in Figure 1 without modifications.

Supermon is a tool for scalable and high speed cluster monitoring. It is split into three separate components, a kernel module for providing the monitored data on each node, a server daemon on each node, and a data aggregator to present a single cluster image to clients. It relies on symbolic expressions for communication between components in order to reduce node perturbation when parsing messages [6]. One drawback to Supermon is that it does not provide memory usage information or node state information. It also does not interface with a cluster scheduler.

Ganglia is a scalable distributed monitoring system targeted towards clusters, grids, and planetary-scale systems [3]. It relies on a multicast-based listen/announce protocol to monitor cluster-wide state information While this approach allows easy setup, it imposes the requirement of having a functional local-area IP multicast network in place. Unlike Fountain, Ganglia is not explicitly designed to interface with a cluster scheduler. However, commercial resource management packages such as Moab[4], or the open source variant Torque[9], can provide interfaces to the data monitored by Ganglia. Ganglia also provides detailed historic usage information by utilizing the RRDtool.

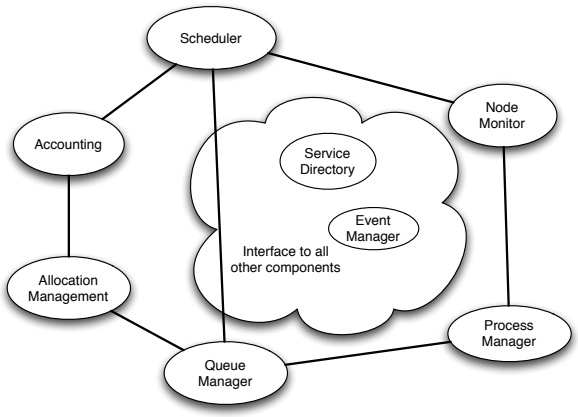

Figure 1. Components of the Scalable Systems Software Project

4 Node Monitor Interface

The primary user of the Fountain node monitoring component is the cluster scheduler. There are other users, mostly for administrative purposes, which will not be discussed further in this paper. The interface Fountain presents to its outside users is fairly straightforward. The system wide Service Directory component handles registration and de-registration requests for all components of the Scalable Systems Software environment. After a component registers itself with the Service Directory, other components can query the Service Directory and ask which port and what protocol a particular component has registered. Upon initializing itself, Fountain registers a port (ex: 9100) and a protocol (ex: SSSRMAP) with the Service Directory.

The Scalable Systems Software Resource Management and Accounting (SSSRMAP) message format defines a request-response syntax based on top of the HTTP protocol [1]. It is suitable for a connection oriented, XML based, application layer client-server protocol for interaction with other SSS components. Fountain uses the SSSRMAP message format and wire protocol, together with the SSS Node Object specification to create its interface to other components. Figure 1 shows some of the other components and their common

```
<Envelope><Body actor="samm">
<Request action="Query"><Data>
</Data><Object>Node</Object>
<Get name="NodeId"></Get>
<Get name="NodeState"></Get>
<Where name="NodeState" op="eq">down
</Where></Request></Body></Envelope>

<Envelope><Body actor="root">
<Response action="Query"><Count>
2</Count><Total>34</Total>
<Data name="NodeList" type="xml">
<Node><NodeId>m20</NodeId>
<NodeState>down</NodeState></Node>
<Node><NodeId>m34</NodeId>
<NodeState>down</NodeState></Node>
</Data><Status><Value>Success</Value>
<Code>000</Code><Message>
2 node(s) found</Message></Status>
</Response></Body></Envelope>
```

Figure 2. Example SSSRMAP node monitor query and response

interactions with one another in the SSS environment.

Fountain is very extensible with respect to the data it can return in response to a query. When any client queries the Fountain server, they can supply an optional `Where` element and indicate an operator such as `gt`, `lt`, `eq`, `ne`, `le`, `ge`, or `like` for regular expression matching. In this fashion the cluster scheduler can query the Fountain server for all nodes matching the particular parameters requested by a user for their parallel job. It is also useful for a system administrator to query the Fountain server for all nodes with a state of `down`.

A sample SSSRMAP node query request and response message is shown in Figure 2. The query asks Fountain to return a response with the nodes that have a state of `down`. The response message indicates there are 2 out of 34 nodes with a state of `down`.

5 Design of Fountain

Fountain consists of three separate components, the Fountain server, the master Fountain daemon, and the slave Fountain daemons. Each component is a separate process that communicates with the other Fountain components using XML messages over sockets. The Fountain server is responsible for aggregating together all of the current node statistical information and making it available to other SSS components using the interface described in the previous section. The master Fountain daemon is responsible for maintaining an accurate topology of slave Fountain daemons. The slave Fountain daemons do the actual monitoring work, they are responsible for monitoring their specific node in the cluster and promptly reporting neighboring Fountain daemon failures. The master and slave Fountain processes are daemons in the sense that they are expected to run forever with little or no user interaction.

A Fountain system starts life by executing the master Fountain daemon and the Fountain server on the head node of a cluster. The second step is to execute a slave Fountain daemon on all the other nodes in the cluster. This step does not need to be fast since it only happens once. Typically the best method to accomplish this is to launch a slave Fountain daemon with an initialization script during the node boot process. For smaller clusters the slave daemons can be launched by hand using ssh, telnet, or a similar method.

5.1 Slave Fountain Daemon

A slave Fountain daemon is very simple. It consists of a process running on each node of a cluster with two purposes. The first is to collect the static and dynamic usage information for the node it is running on, and the second is to report neighboring Fountain daemon failures in a timely fashion. In the current implementation, Linux is the only supported architecture for the slave Fountain daemons because the required monitoring information can be found in the `/proc` file system. The interface to gather the monitoring information is abstract enough that it will be trivial for Fountain to support other operating systems in the future. The information collected by each daemon is the amount of configured memory and swap space, amount of available memory and swap space, CPU usage, CPU architecture, and node operating system.

When a slave Fountain daemon starts, it performs some initialization work that opens a listening socket for incoming connections. Then it attempts to connect to the master Fountain daemon, after which it enters what is essentially an infinite loop. Inside this infinite loop, a system call to `select` waits for an incoming connection on the listener socket or for incoming data on an existing socket. If there are no incoming connection requests, or data waiting to be read on existing sockets, the slave Fountain daemons essentially do nothing. They only collect node information when requested to. That is, the Fountain server pulls data from the Fountain daemons rather than the daemons pushing the data.

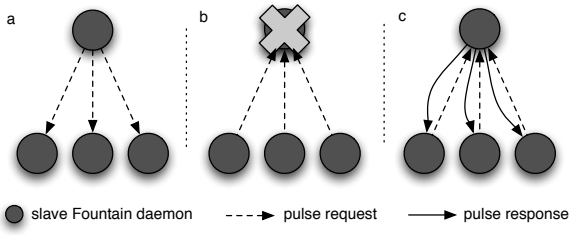

Figure 3. The three purposes of the pulse request and response messages.

A slave Fountain daemon has a persistent connection to its parent node and persistent connections to any number of child nodes. The number of child connections for each Fountain daemon depends on its location in the Fountain tree topology, which will be explained in the next section. Slave Fountain daemons respond to request messages received from their parent node. An example request is a `query` message, which instructs the daemon to gather its node statistical information. Slave Fountain daemons also expect to receive two types of messages from their child nodes, a pulse request or a zero length message. If a zero length message is received from either the parent connection or any of the child connections, the slave Fountain daemon assumes that neighboring node has failed and it reports the failure to the master Fountain daemon.

Pulse messages serve three purposes, they are shown in Figure 3. The first purpose, in Figure 3a, happens when a parent daemon periodically sends its children a pulse request. When a child receives this message, it knows the parent is alive and well. The second pulse message purpose is shown in Figure 3b, when a parent daemon dies unexpectedly and the remote socket is not closed. This could happen due to a kernel panic, an unplugged network cable, or any number of other reasons. In this case, the child will send a pulse request to its parent. If it does not receive a response, the parent is assumed to be lost and handled as previously mentioned. The third purpose is shown in Figure 3c, when the parent daemon has not sent its children a message during a predefined time interval but it's still alive. In this case, the children will send a pulse request to their parent and it will respond with a pulse response indicating it's still alive.

5.2 Master Fountain Daemon

A Fountain system consists of a single master Fountain daemon, typically running on the head node of a cluster. The master daemon is similar to the slave daemons, except it has the added requirement of maintaining an accurate topology of slave Fountain daemons in order to facilitate fast node queries. To achieve maximum scalability, we chose to use an n-ary tree topology to manage the slave Fountain daemons instead of other topologies such as a ring or a star [2]. Figure 4 shows an example ternary Fountain tree topology. Section 6 will present some initial conclusions about the optimal number of children for each node based on node query performance results.

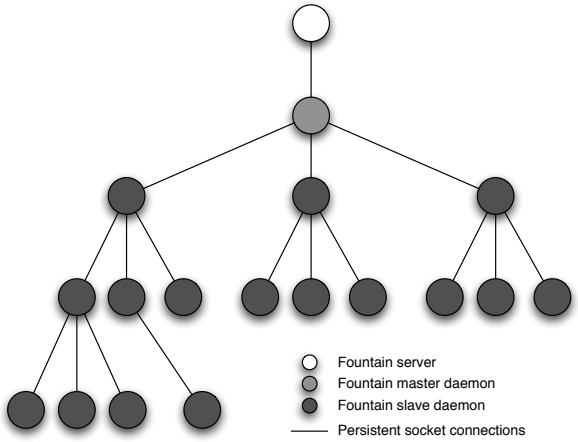

Figure 4. Example Fountain tree topology using 3 children per node

When the master Fountain daemon starts, it opens a listening socket and waits for incoming connections. When a slave Fountain daemon connects to the master daemon, they initiate the tree establishment algorithm. This algorithm uses a three way handshake to introduce new slave Fountain nodes daemons the tree topology. After receiving the initial connection request, the master daemon looks up the next available parent daemon in its tree topology data structure. An available parent node is defined as the first node in the topology with less than the maximum number of children. The master daemon then sends a join response containing the hostname and listening port of the parent daemon this slave daemon should connect to.

After receiving the join response, the slave Fountain daemon attempts to join the parent daemon specified by the master daemon. When this connection is successful, the slave daemon responds to both its new parent daemon and the master daemon with a join-ack response and closes its connection to the master Fountain daemon since its no longer necessary. The master Fountain daemon then appends the newly joined slave daemon to its tree topology data structure and increments the number of children for its parent daemon

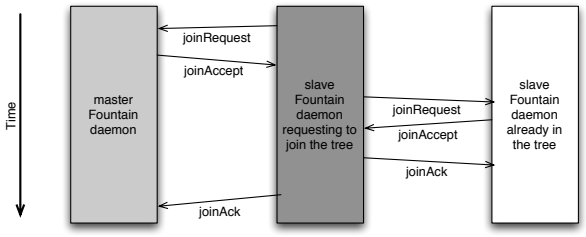

Figure 5. Fountain tree establishment message sequence

by one. The sequence of messages exchanged between the master Fountain daemon, a slave Fountain daemon already in the tree topology, and a slave Fountain daemon that is attempting to join the tree topology is shown in Figure 5.

The Fountain tree topology is designed as a complete n-ary topology. This means each entry has at most `n` children, and all the levels of the tree are full except for the bottom level, which is filled from left to right. Equations 1 and 2 can be used to locate the parent of node `b`, or child `a` of node `b`. The result of each equation is an index into the tree topology data structure. Note that `a` can have values between `0` and `n-1` inclusive.

$$parent = \lfloor (b-1)/n \rfloor \tag{1}$$

$$child = n * b + a \tag{2}$$

In addition to the tree establishment algorithm, the master Fountain daemon uses two additional algorithms to maintain the the topology of slave Fountain daemons. The tree recovery and rebuilding algorithms are used to recover the tree topology in the event of node failures. In this context, a node failure is defined as an event that causes the slave Fountain daemon to not respond to request messages. How the node failure happens is not necessarily important, just that the system of Fountain daemons can handle such an event.

The tree recovery algorithm used by Fountain is based on the work in [2]. When a slave Fountain daemon in the tree topology fails, both its parent daemon and child daemons notice the failure when their socket connection to that daemon is closed unexpectedly. They will then attempt to report this daemon failure to the master Fountain daemon. Upon receiving a lost daemon request, the master daemon will transition the tree state from idle to recovery. Once in the recovery state, the master daemon rejects all other requests except for additional lost daemon requests. That is, when the Fountain server sends a node query to the master daemon while the tree topology is in recovery state, it will receive an error response indicating the tree topology is recovering from a failure. Requests to join the tree topology are also rejected by the master Fountain daemon when the tree topology is recovering. After initially entering the tree topology recovery state, the master daemon marks the failed daemon as lost and waits for all of the lost daemon's neighbors to report the failure. After all of the failed daemon's neighbors have contacted the master Fountain daemon to report the failure, the master daemon selects a replacement daemon from the tree topology. To minimize the number of slave Fountain daemons affected by this recovery algorithm, the replacement daemon is always the last Fountain daemon to join the tree topology. After the replacement daemon successfully joins the failed daemon's parent, the master Fountain daemon informs the failed daemon's children to join the replacement daemon. The master daemon then sets the tree state back to idle so query requests can be processed.

If multiple nodes fail at or near the same time, the master Fountain daemon will accept the first lost node request and reject all subsequent requests by informing the daemon that reported the failure to try again later. Clearly this could cause a race condition if a single slave daemon has both its parent and one or more of its children fail concurrently. This race condition is mitigated by using a timer when the master Fountain daemon initially transitions to the tree topology recovery state. If the timer expires before all of the neighboring Fountain daemons have reported the failure, the master daemon transitions from the tree topology recovery state to the tree topology rebuild state. Once in the rebuild state, the master daemon closes all of its child connection sockets and responds to all lost node requests by telling the slave Fountain daemons reporting node failures to rejoin the master Fountain daemon. After receiving a rejoin response when reporting a lost node request, a slave Fountain daemon will close all of its child connections and attempt to rejoin the master Fountain daemon. This process happens recursively until the entire tree topology is rebuilt.

The tree recovery algorithm is shown in Figure 6 in the event of a single node failure where each node in the tree has a maximum of three child nodes. The resulting tree topology after the recovery would have node 4 replaced by node 16. Nodes 13, 14, and 15 would be children of node 16.

Up until this point, we have only discussed failures of the slave Fountain daemons. It is possible for the master Fountain daemon to fail as well. This will be detected by the master daemon's direct children. Since each slave daemon has no knowledge if its parent is the

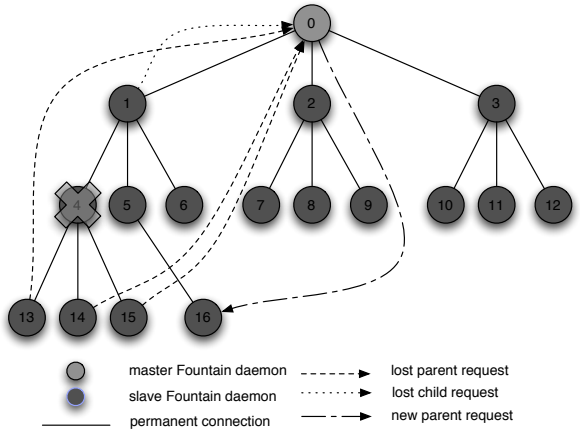

Figure 6. Fountain tree topology recovering from a single node failure

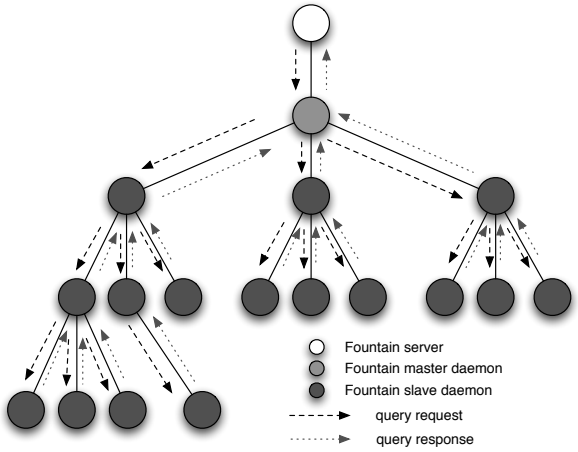

Figure 7. Query request & response

master daemon or another slave daemon, they attempt to connect to the master daemon and report its own failure. Obviously this will not succeed since the master daemon has failed and cannot accept connections. When the master daemon eventually returns, its former children will connect successfully and attempt to report the failure. The master daemon responds to this request by informing the slave daemon to rejoin the tree topology. This process is essentially the same as the tree rebuilding algorithm. After being told to rejoin the tree, the slave daemons that reported the master daemon failure will rejoin the master daemon and close their child connections. Their children, in turn will attempt to report their failure, which will cause the master daemon to inform them to rejoin the tree.

5.3 Fountain Server

The Fountain server is the most important component of the Fountain node monitoring system since it presents a single system view of the cluster to clients. After starting the Fountain server, it opens a listening socket for client requests and enters its main loop. Inside the main loop it attempts to connect to the master Fountain daemon. This connection is persistent, so it only happens once. After successfully connecting, it sends the master Fountain daemon a query request message at a user configurable interval. After sending the query request message, the Fountain server waits for the query response from the master Fountain daemon. This query response will contain the node status information from each node in the Fountain tree topology. The Fountain server maintains a node monitor data structure to keep track of all the information for each node in the cluster. This data structure is updated after receiving and parsing a node query response. The node query request and response message sequence between the Fountain server, and all of the Fountain daemons is shown in Figure 7. Also inside its main loop, the Fountain server accepts incoming client connections and responds to their requests. These connections are transient, so they are closed after sending the response message.

Since the primary client of the Fountain server is the cluster scheduler, accurate node state information is imperative for the scheduler to operate effectively. To fulfill this requirement, Fountain will change a node's state to `down` if it does not respond to a query request. Only after the node comes back online and the Fountain daemon for that node responds to a query request, will the node's state be changed back to `Up`.

When the master Fountain daemon receives the query request from the Fountain server it immediately forwards the request to each of its children and then waits to receive a response from them. This process happens recursively for each slave Fountain daemon as well. The slave Fountain daemons that are the leaf nodes of the tree topology have no children, and will immediately respond to the query request with their node status information. When their parent daemons receive a query response from each of their child daemons, they append their node status information together with each child response and send the response to their parent daemon. When the master Fountain daemon receives query response messages from each of its children, it appends its node status information and sends the response to the Fountain server.

A race condition exists if a node in the cluster fails

before the query request message has reached the slave Fountain daemon running on that node. This happens due to the use of a blocking read after the master Fountain daemon sends the query request message to its children. When a slave Fountain daemon in the tree topology fails, its neighboring daemons will attempt to report this failure to the master Fountain daemon. Since the master daemon is a single threaded application, it cannot respond to the node failure request and wait for the node query response at the same time. This race condition is handled by the addition of a timeout when the master daemon is waiting for query response messages from each of its children. If the timeout period has elapsed before receiving a query response from one of its children, the master Fountain daemon will abort the node query request and send a response with a failure error code to the Fountain server. Then it will handle the node failure request and recovery the tree topology using the tree recovery algorithm.

6 Results

In this section we present test results from two medium sized Linux clusters. The first test environment is 4pack, a cluster of 34 PowerPC G4 Macintosh computers running Debian Linux and connected by a high speed Myrinet network for intra-node communication and fast ethernet for management. The second test environment is Scink, a 64 node dual-processor AMD Athlon MP2200 cluster running Debian Linux and connected with a 2D SCI network and fast ethernet.

6.1 Query Performance

The primary design goal of Fountain is to maintain an accurate tree topology in the presence of node failures. Performing fast and efficient node queries is a secondary goal. To do this the Fountain daemons are arranged in a n-ary tree topology as described previously. The results from performing node queries on a variety of Fountain configurations are shown in Tables 1 and 2. Figures 8 and 9 show a graph of these two tables. To achieve larger configurations than 34 Fountain daemons on 4pack or 64 Fountain daemons on Scink, multiple slave Fountain daemons were run on a each node in the cluster. The time represented for each query is measured as the time it takes the Fountain server to send the master Fountain node a query message, receive the query response, and process the response message. They are measured in milliseconds and represent an average of three separate node queries.

Table 1. Elapsed node query time on 4pack (milliseconds)

System Size	Binary	Ternary	4-ary	5-ary
34	140.07	97.32	95.8	86.01
67	189.88	180.04	154.85	157.09
100	289.28	213.05	225.32	212.24
133	370.07	308.05	304.31	225.34
199	559.38	478.68	407.06	417.11
265	675.85	564.34	488.31	552.37

Table 2. Elapsed node query time on Scink (milliseconds)

System Size	Binary	Ternary	4-ary	5-ary
65	106.92	125.31	93.86	98.10
129	195.65	207.85	151.32	147.13
257	443.78	462.12	331.67	267.52
512	721.96	682.11	543.56	550.49
769	985.29	1067.4	901.45	607.83
1025	1112.15	1034.68	972.51	898.02

On both 4pack and Scink the results show nearly linear scaling. More-so on 4pack than Scink, which is probably because Scink is used as a production cluster for computational chemists so CPU usage was not as low as 4pack when these test results were collected. The query time results are favorable in the sense they show Fountain is capable of scaling to larger system configurations without adversely affecting the time it takes to query every node in the cluster. These results show that Fountain is capable of querying and processing the results for a cluster with up to 1,025 nodes in less than one second. If faster monitoring speed is desirable for a particular installation site, Fountain can be extended to act as a wrapper around a more specialized monitoring component like Supermon [6] and present its data using the SSS interface.

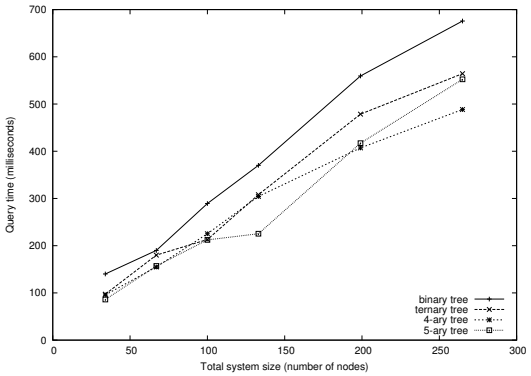

Figure 8. Elapsed time to perform a node query on 4pack

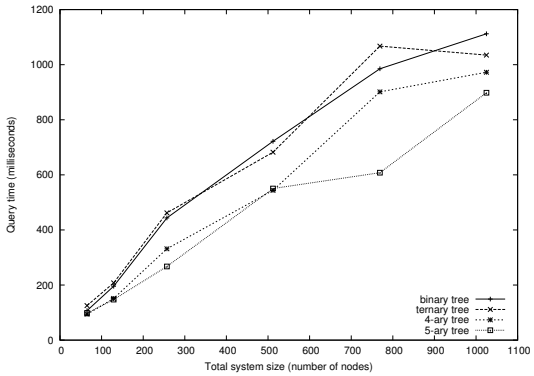

Figure 9. Elapsed time to perform a node query on Scink

6.2 Recovery Performance

The 4-ary and 5-ary tree topologies perform node queries faster than the binary and ternary trees. However, with these configurations the time for the tree to recover from a single node failure increases. Table 3 shows the average time to recover the tree topology from a single deterministic node failure. The numbers shown represent the time in milliseconds that elapse from when a neighboring node first reports the failure until a replacement node has been contacted and successfully replaced the failed node. For tree topologies with a larger degree, the time to recover from a node failure is longer because each node has more neighbors that have to report the failure. Table 4 shows the average time to recover the tree topology from multiple concurrent node failures on 4pack. In this case, the time shown is the total time that elapses for the tree topology to recover from each failure. The time to recover from multiple concurrent node failures is more than the time to recover from a single node failure, multiplied by the number of failed nodes. Since the master Fountain daemon can only recover the tree topology from a single failure at once, it rejects all lost node requests for a different lost node if it is already handling a node failure. Only after the master daemon has recovered the tree topology from the first failed node that was reported first, will it respond to requests for another lost node.

These recovery times show that Fountain is capable of efficiently reconstructing its tree topology from single or multiple node failures by affecting the minimum number of nodes possible. The recovery results are favorable since the time to recover from both single and multiple node failures only depends on the degree of the tree topology and not the total number of fountain daemons.

Table 3. Elapsed tree topology recovery time from a single deterministic node failure (milliseconds)

System	Binary	Ternary	4-ary	5-ary
4pack	95.23	167.66	195.74	230.76
scink	156.73	195.43	247.28	276.69

6.3 Rebuild Performance

The tree rebuilding algorithm presented in section 5.2 is used as a last effort to reconstruct the tree topology if the master Fountain daemon determines it's not possible to recover the tree topology using the tree recovery algorithm. The numbers shown in Table 5 represent a worst case scenario for the tree rebuilding algorithm. The elapsed time is obtained by forcing a tree rebuild without losing any nodes in the cluster. Normally, the tree rebuilding algorithm would be triggered by multiple nodes failing at the same time. In that case, timing the rebuilding algorithm would be difficult because it's unknown how many nodes failed and when they will be able to rejoin the tree topology. The tree rebuilding algorithm is faster on topology configurations with a higher degree, which is expected due to

Table 4. Elapsed tree topology recovery time from multiple node failures on 4pack (seconds)

# Failed	Binary	Ternary	4-ary	5-ary
2	0.22	0.35	0.49	0.49
3	1.24	1.36	1.05	0.72
4	1.36	2.3	2.40	2.638
5	0.748	1.85	2.66	2.33

Table 5. Elapsed tree topology rebuild time on 4pack (seconds)

System Size	Binary	Ternary	4-ary	5-ary
33	5.54	4.46	3.93	3.77
65	6.70	5.85	4.847	4.98
97	7.87	6.46	6.56	6.01
161	9.94	9.02	8.72	8.61

its design.

6.4 Node Overhead

Table 6 and Figure 10 summarize the node overhead for individual nodes on 4pack. The monitoring bandwidth numbers were collected by running tcpstat [8] three times, for five minutes and averaging the results. CPU usage is not shown in the table because it was less than 0.1% in all cases but the master daemon, where it was 0.15%. Each Fountain daemon achieves low CPU usage because it spends most of its time waiting for incoming messages in the `select` system call.

The bandwidth numbers shown in Table 6 represent both the send and receive bandwidth for each Fountain daemon, depending on the query interval used by the Fountain server, and the daemon's particular level in the tree topology. The level is the number of hops away from the master Fountain daemon. The master daemon uses the most bandwidth since it sends a query response message containing all of the Fountain daemon's information to the Fountain server. As the level increases, the bandwidth usage per node decreases since that node will have at most, half as many children and grandchildren as its parent node. Increasing

Table 6. Node bandwidth (KB/sec) usage on 4pack for a binary tree topology

query interval	level	total nodes	bandwidth
30	0	33	1.3
30	1	33	0.78
30	2	33	0.47
30	0	65	2.42
30	1	65	1.40
30	2	65	0.49
15	0	33	2.60
15	1	33	1.41
15	2	33	0.87
15	0	65	4.8
15	1	65	2.6
15	2	65	1.5

the query interval from 30 seconds to 15 seconds increases the bandwidth by about a factor of 2. This is expected because the query request and response messages make up nearly all of the bandwidth used by a Fountain daemon. The only other messages sent between Fountain daemons are the `pulse` request and response messages, which are very small and sent less frequently than query messages.

The numbers shown represent the bandwidth when the tree topology is in a binary tree configuration. As expected, increasing the tree topology degree causes the bandwidth per node to decrease more rapidly as the level number increases. However, the bandwidth used by the master Fountain daemon will always be the same as long as the total number of Fountain daemons does not change.

7 Future Work

Due to its fault tolerant design requirement, additional work remains to formally verify the tree establishment, tree recovery, and tree rebuilding algorithms used to maintain the tree topology for the slave Fountain daemons. It may also be beneficial to implement a more advanced algorithm for establishing the tree topology where the number of children each Fountain daemon has depends on its depth in the topology instead of using a fixed number of children per daemon.

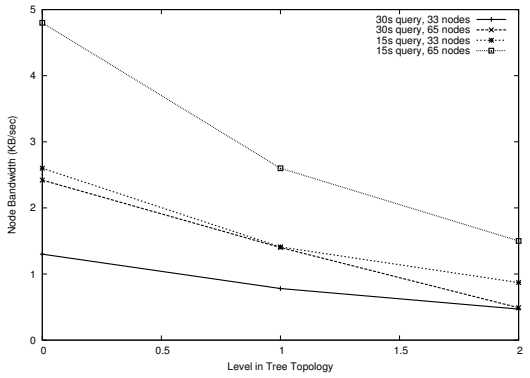

Figure 10. Node bandwidth usage on 4pack for a binary tree topology

Extending the Fountain server to monitor other data than just node specific information is also a future research topic. We have explored this topic by implementing an Infiniband network monitoring module and have considered extending this idea to monitor other high performance networks. This information is gathered by the Fountain server separately from the node based information from the Fountain node daemons. This idea could also be implemented to monitor specialized parallel architectures such as the Cray XT3 or IBM BlueGene. Finally, a server specific data source could act as a wrapper around other, more specialized monitoring tools, such as Ganglia, Supermon, or NWPerf [5], and allow Fountain to present their data to other SSS components using the existing interface to Fountain.

To provide a user friendly interface to Fountain, both a web based and traditional GUI application are in development. These projects will allow a system administrator to view the overall cluster status as a snapshot. Historical system usage network topography visualization are being considered for this project.

8 Conclusion

This paper has described the design of a node monitoring component that implements the Scalable Systems Software node monitor specification. The results show it scales favorably to larger systems sizes when performing node queries and recovering from node failures. Future work exists to add additional features such as a web interface and network topology performance mapping.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the Mathematical, Information and Computational Science division of the Office of Advanced Scientific Computing research.

References

[1] B. Bode and S. Jackson. Scalable systems software resource management and accounting documentation. `http://sss.scl.ameslab.gov/docs.shtml`, 2005. [Online; accessed April 5, 2006].

[2] M. Huang and B. Bode. A performance comparison of tree and ring topologies in distributed systems. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, pages 258–266, Denver, Colorado, Apr. 2005.

[3] M. L. Massie. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30:817–840, July 2004.

[4] Moab cluster suite. `http://www.clusterresources.com/pages/products/moab-cluster-suite.php`, 2006. [Online; accessed April 12, 2006].

[5] R. S. R. Mooney, K.P. Schmidt. Nwperf: a system wide performance monitoring tool for large linux clusters. In *Proc. IEEE International Conference on Cluster Computing*, pages 379–389, Denver, Colorado, Sept. 2004.

[6] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proc. IEEE International Conference on Cluster Computing)*, pages 39–46, Chicago, Illinois, Sept. 2002.

[7] The scalable systems software website. `http://www.scidac.org/scalablesystems`, 2005. [Online; accessed November 8, 2005].

[8] tcpstat. `http://www.frenchfries.net/paul/tcpstat/`, 2006. [Online; accessed April 10, 2006].

[9] Torque resource manager. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`, 2006. [Online; accessed April 12, 2006].