The node monitoring component of a scalable systems software environment

by

Samuel James Miller

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Brett Bode, Co-major Professor
Srinivas Aluru, Co-major Professor
Robyn Lutz

Iowa State University

Ames, Iowa

2006

Copyright © Samuel James Miller, 2006. All rights reserved.

Graduate College
Iowa State University

This is to certify that the master's thesis of

Samuel James Miller

has met the thesis requirements of Iowa State University

Co-major Professor

Co-major Professor

For the Major Program

DEDICATION

I would like to dedicate this thesis to my wife Erin, without her support I would not have been able to complete this work. I would also like to thank my parents and brother for their love and support while I have been in school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This research describes Fountain, a suite of software used to monitor the resources of a cluster. A cluster is a collection of individual computers that are connected with a high speed communication network. They are traditionally used by users who desire more resources, such as processing power and memory, than any single computer can provide. A common drawback to effectively utilizing such a large scale system is the management infrastructure, which often does not often scale well as the system grows.

Large-scale parallel systems provide new research challenges in the area of systems software, the programs or tools that manage the system from boot-up to running a parallel job. The approach presented in this thesis utilizes a collection separate components that communicate together to achieve a common goal. While systems software comprises a broad array of components, this thesis focuses on the design choices for a node monitoring component. We will describe Fountain, an implementation of the Scalable Systems Software (SSS) node monitor specification. It is targeted at aggregate node monitoring for clusters, focusing on both scalability and fault tolerance as its design goals. It leverages widely used technologies such as XML and HTTP to present an interface to other components in the SSS environment.

CHAPTER 1. RESEARCH BACKGROUND

1.1 Scalable Systems Software

Over the past few years, the nation's premiere scientific computing centers have continued to expand the size of their high-end parallel systems into the multi-teraflop domain in order to solve complex problems in areas such as life sciencies, nuclear reactions, and global climate simulations. Today, there are many systems with thousands of processors and some approaching over one hundred thousand processors. This exponential growth on the hardware end has been largely hindered by the lack of scalability of software used to manage these parallel systems. The computer industry is not motivated to solve this problem since the market pushes them towards smaller systems targeted at businesses and departmental systems. To mitigate this problem, the U.S. Department of Energy has established the Scalable Systems Software (SSS) Center.

The goal of the SSS center is to develop a suite of machine independent, component based, open source cluster management software to more effectively manage and utilize computational resources. The desire for such software comes from the Scientific Discovery through Advanced Computing (SciDAC) initiative [28]. As computational resources grow beyond today's teraflop class systems into future petaflop and beyond systems, scalability problems must be solved to utilize these resources to their full potential. The systems software problems for such systems are significantly more complex than smaller scale systems. Issues such as fault-tolerance, reliability, manageability, and ease of use for system administrators and users all present significant research challenges for large scale systems that are not seen in smaller systems.

A key goal of the SSS project is its modularity by using a component based design. The requirement for individual components to use a publicly documented interface allows for instal-

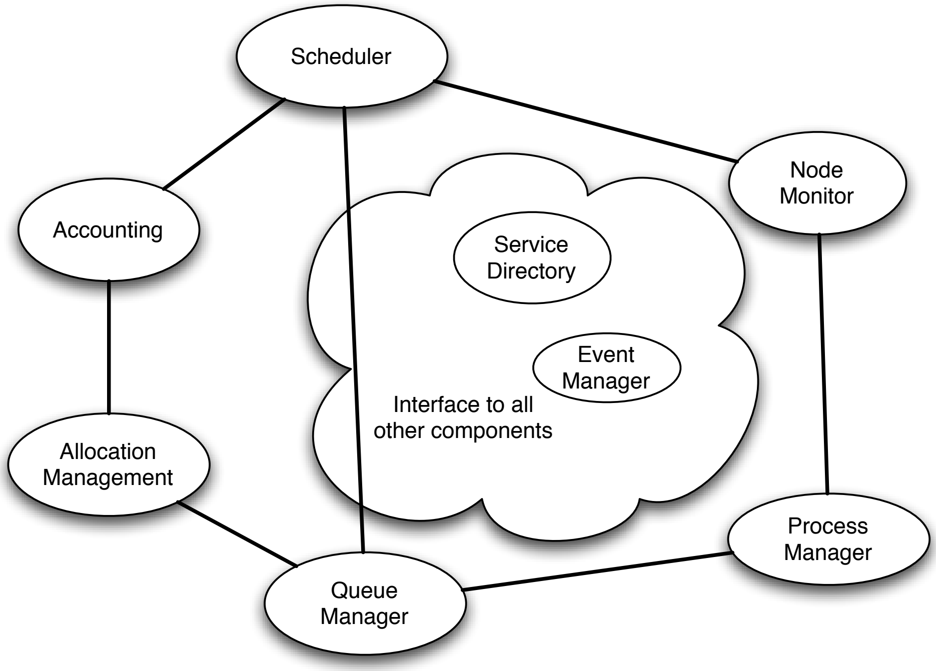


Figure 1.1 Components of the Scalable Systems Software Project

lation sites to substitute a specialized component if desired. This goal was anticipated because next generation high-end parallel systems will be very diverse and often require specialized components to manage their resources effectively.

Figure 1.1 shows an example SSS installation with a few of the components. The system wide service directory and event manager component shown in the middle connect to all the other components and allow them to announce their participation in the system. They provide facilities to communicate both synchronously in the form of send and receive methods, and asynchronously in the form of registration and notification events.

1.2 Motivation and Problem Statement

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. A typical sequence of events for a primitive cluster management infrastructure is as follows:

1. User notices his job has not completed as expected, notifies system admin
2. System admin tries to ssh or telnet to the node
3. This may or may not succeed, depending on the node's status
4. If the login fails, the system admin pings the node
5. If the ping fails, the system admin reboots the node

This methodology is inefficient and excessively time consuming for even medium sized clusters. As clusters continue to increase in size, effective monitoring of system and node information will be required to prevent loosing both valuable CPU hours, and the time spent by system administrators fixing problems.

Scalability issues with both the performance and overhead of a monitoring system are more prominent as cluster sizes increase into the thousand node domain. Traditional resource management systems such as the Portable Batch System (PBS) [22] have utilized transient connections to both start parallel jobs and monitor resource usage. This methodology is well understood and quite effective for smaller cluster sizes on the order of tens or hundreds of nodes, but the cost of connection setup and tear down for a cluster with thousands of nodes is too great. A rigid topology of persistent daemons running on each compute node that is capable of withstanding individual and multiple node failures would be more effective for larger clusters because the connection setup overhead amortized over the lifetime of the topology. With a rigid topology, messages can traverse the network much faster since the connections are persistent.

User jobs can be started in a nearly parallel fashion, and resources can be monitored much more effectively than with transient connections. The disadvantage of a rigid topology is it has to be reconstructed or recovered in the event of node failures, which can be a time consuming process depending on the topology selected.

To solve these problems, we have created a node monitoring tool called Fountain that implements the Scalable Systems Software node monitor specification. Fountain has three distinct design goals in order to be a reliable, accurate, and scalable 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 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. Scalability in this sense, means Fountain should not encounter adverse performance issues when it's installed on larger parallel systems, such as exhausting the operating system's file descriptors.

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 such a manor that does not adversely affect the system.

1.2.1 Node Monitoring

Monitoring is the act of observing a system via a set of sensors, through either periodic or reactive means. Periodic monitoring samples the system at regular intervals while reactive monitoring samples the system as a result of an external event, typically a failure or warning.

Typically, the desire for monitoring a system stems from the management infrastructure. When the sensor values are read, a more informed decision can be made about what actions to take. For example, a modern day car typically has an oil pressure light in addition to a gauge that displays the current oil pressure. If that pressure ever drops below a certain level, the light alerts the driver that a problem exists and it should be fixed by a qualified mechanic before a more extensive problem arises. This example can be extended into computers as well, with sensors capable of monitoring environmental metrics such as cpu temperature, ambient air temperature, and fan speed. As well as performance metrics such as CPU utilization, memory bandwidth, and network performance.

The role of a node monitor in a scalable systems software environment is to collect relevant information from each node in the cluster. As mentioned in the previous section, the minimal information collected should be the status of each node, so we can provide the cluster scheduler with a basic idea of which nodes are available to run user jobs. Additional information collected can vary between installation sites and the components utilized. Common metrics include percentage of CPU utilization, available memory, amount of local disk space available, and available network connectivity. Other proactive metrics, such as power supply status, fan speeds, CPU temperature(s), and hard drive status may also be desired to detect failures before they happen. How the information is collected is an important research topic that will be explained in the following chapters.

The node monitoring component plays a critical role when scheduling user's jobs. When the cluster scheduler needs to run a job, it will query the node monitor to determine what resources are available and compare those that the job has requested. To achieve a high degree of utilization for a particular cluster, the scheduler needs to have an accurate snapshot of the cluster's status.

1.3 Summary

The Scalable Systems Software project exists to provide the nation's premiere scientific computing centers with component based, open source software to more effectively utilize and manage both their small and large scale parallel systems. By providing supercomputer centers such an interoperable framework of components to manage their clusters, it will allow them to easily adapt, update, and maintain their components in order to keep up with new hardware and software. We developed the Fountain node monitor to implement the SSS node monitor specification and provide other SSS components with the relevant information they need to accomplish their goals.

CHAPTER 2. LITERATURE REVIEW

2.1 Monolithic Systems

Many monolithic resource management systems are available to manage both small and large scale parallel systems. Typically these systems provide services for scheduling user jobs, monitoring the node and job status, launching user processes, and managing the cluster's configuration. The Portable Batch System [22] is a flexible resource management system for clusters. It has been used successfully on numerous smaller clusters, but it suffers from scalability issues on larger configurations. The Simple Linux Utility for Resource Management (SLURM) is a resource management system developed at Lawrence Livermore National Laboratory [13]. It is open source and highly portable across many UNIX-like operating systems. It features a novel plug-in architecture allowing a great deal of extensibility for the user. Both IBM's LoadLeveler [15] and Platform Computing's Load Sharing Facility (LSF) [16] are commercial resource management packages that provide similar facilities.

Many of these products encompass all of the functionality of the various SSS components (see figure 1.1) into one or a few components. Without modifications, these resource management systems are not suitable to operate in an SSS environment due to their monolithic nature.

2.2 Dedicated Node Monitors

In addition to the monolithic resource management systems presented above, there exist several commercial and open source packages that focus solely on the node monitoring aspect of a parallel system. These tools will be outlined and their benefits and shortcomings will be discussed as relevant to the SSS project.

2.2.1 Ganglia

The Ganglia distributed monitoring system is a scalable monitoring system targeted towards clusters, grids, and planetary-scale systems [17]. It relies on a multicast-based listen/announce protocol to monitor cluster-wide state information. To promote scalability it utilizes a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state into a single system image, see Figure 2.1. Each node in a ganglia system monitors its local resources and sends multicast packets containing this data whenever significant updates occur. The threshold for publishing monitoring data is user configurable depending on which metric is monitored.

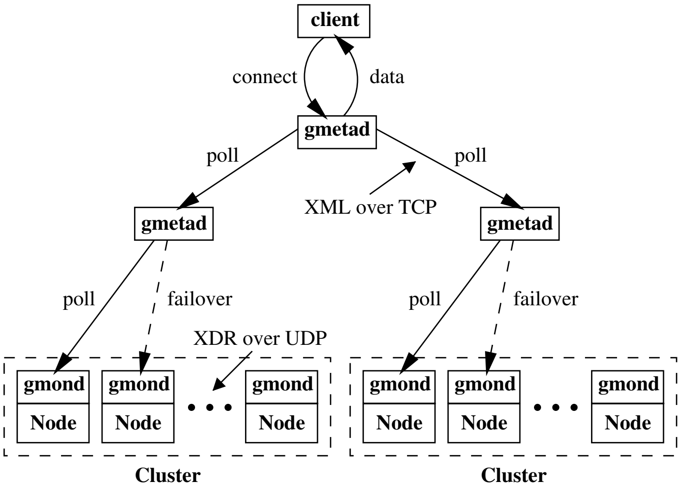


Figure 2.1 Ganglia architecture. Figure taken from [17]

The use of a multicast-based listen/announce protocol has several advantages for large scale distributed systems. First, it requires no manual configuration of a node list or topology. Secondly, it allows any node in the cluster to be aware of the cluster's state at any time, which promotes excellent redundancy given the increased failure rate as cluster sizes grow larger. However there are some drawbacks to using such a method of data aggregation. Such a network may not always be feasible, especially for larger cluster configurations. The authors conclude that even with a reduction in monitoring frequency, a cluster of 2000 nodes exhibits a multicast packet rate of 813 packets per second [17]. Clearly Ganglia faces some scalability challenges if the design choice of a local-area multicast network is used as the primary means for collecting monitoring data in future versions.

Unlike Fountain, Ganglia is not explicitly designed to interface with a cluster scheduler. However, commercial resource management packages such as Moab[20], or the open source variant Torque[32], can provide interfaces to the data monitored by Ganglia. Ganglia is highly portable and currently used on over 500 clusters around the world [17]. It also provides a nice web-based front end tool to monitor a cluster's state, as well as view detailed historic usage information by utilizing the RRDtool.

2.2.2 Supermon

Supermon is a set of tools 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 at all levels for communication between components in order to reduce node perturbation when parsing messages [27]. A single *mon* process runs on all the compute nodes, they parse the symbolic expressions from the kernel module and make that information available over a TCP port to clients and the *supermon* data aggregator. A *supermon* data aggregator connects to any number of *mon* processes using their TCP ports and queries them for data in a linear fashion. Supermon achieves scalability to larger cluster configurations by using multiple data aggregators connected in a tree topology, shown in Figure 2.2.

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. These drawbacks are due to Supermon's design goal of high efficiency and raw monitoring speed. While their extremely high monitoring speed may be useful in cases where fine grained behavior is desirable, its novelty falls outside the scope of the SSS project in terms of a node monitor.

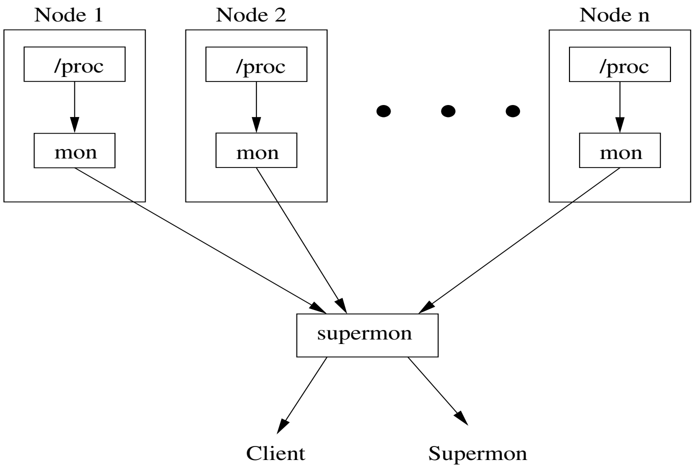


Figure 2.2 Supermon topology. Figure taken from [27]

2.2.3 NWPerf

The NWPerf toolkit is designed to allow metrics from a cluster to be gathered for large configurations in a low-impact and high resolution manner [23]. The primary goal of the toolkit is to monitor machine behavior in the presence of user applications. Their design consists of a multi-tiered architecture where a lightweight client performs the monitoring work, a server side packet handler handles the monitoring packets from each client, and an external storage system maintains historic information. Analysis and reporting tools can be built on top of the storage system to allow further processing of the monitored data. The monitoring metrics collected by a NWPerf system include, floating point operations per second, memory bandwidth, virtual memory statistics, cpu usage statistics, and local filesystem statistics.

The lightweight clients that run on each node use a multicast transport protocol in favor of a unicast transport protocol to allow the future addition of multiple data collectors. In its present implementation, the server only runs a database collector. Future additions could include a real-time visualization tool or similar monitoring components. The use of UDP multicast packets also greatly simplifies the lightweight client application on each node, since the socket code is much more concise than a comparable unicast implementation. However, the authors conceded that care must be taken to ensure reliable delivery of the packets since the network stack may discard them if the server packet handler does not read them in time.

Minimizing monitoring perturbation is a key goal of the NWPerf project. They achieve this by minimizing the interval at which the NWPerf clients execute across all of the compute nodes. The monitoring data collection is scheduled to happen during a short interval, at which point each client gathers its data and transmits it to the server. To accomplish this synchronization, all of the nodes in a cluster are time synchronized using a network time protocol (NTP) server. The authors of NWPerf acknowledge some deficiencies in this solution, particularly clock skew after a node outage, and propose using a central server to trigger data collection.

Similar to Supermon, NWPerf is targeted more at analysis of parallel job performance instead of providing node state information to a cluster scheduler. Its novelty lies in its ability collect monitoring data for large clusters with relatively small perturbation.

2.3 Parallel Process Management

Much of our research inspiration comes from the desire to expand upon the functionality provided by the MPD parallel process management system [7]. While the purposes of MPD and Fountain are inherently different, they both share a goal of having a scalable topology of daemons running on each compute node of a cluster. The topology chosen by MPD is a ring topology (figure 2.3), where each MPD daemon is connected to both a left neighbor and right neighbor. Like Fountain, MPD employs a single daemon per host in a TCP-connected network. Their daemons are persistent and expected to run for weeks, months, or as long as system uptime is feasible.

Three important goals of the MPD project are to have fast job startup, achieve reasonable scalability for large cluster installations, and be somewhat robust. The goal of robustness comes from the desire to tolerate higher failure rate as cluster sizes increase. At any size, an unexpected failure of any part of the system should not bring MPD down. In that sense, their goal of robustness maps directly to a ring topology where there is no master or root process. The goal of fast job startup and scalability also promote using such a topology. As a message traverse the ring, user processes can be forked off nearly in parallel. Even though

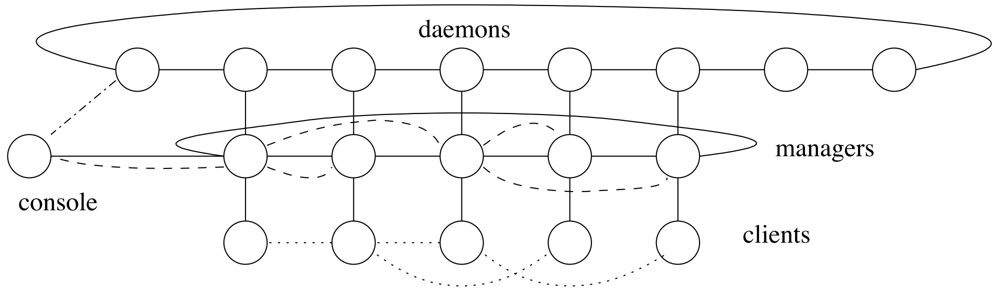


Figure 2.3 MPD daemons with console process, managers, and clients. Figure taken from [7]

such a topology is not as scalable as others, the author's experiments show it is feasible for the thousand-daemon domain [7].

CHAPTER 3. SYSTEM STRUCTURE

This chapter outlines the basic structure, technical requirements, and design choices for Fountain.

3.1 Introduction

Fountain consists of four separate components shown in Figure 3.1, the Fountain server, the master Fountain daemon, the slave Fountain daemons, and the Fountain client utilities. Each component exists as a separate process that communicates with the other components using XML messages over TCP sockets. The Fountain server and master daemon are typically run on the head node of a cluster, while the slave daemons run on all of the compute nodes. The tree topology used to arrange the slave daemons shown in Figure 3.1 will be explained in section 4.2.

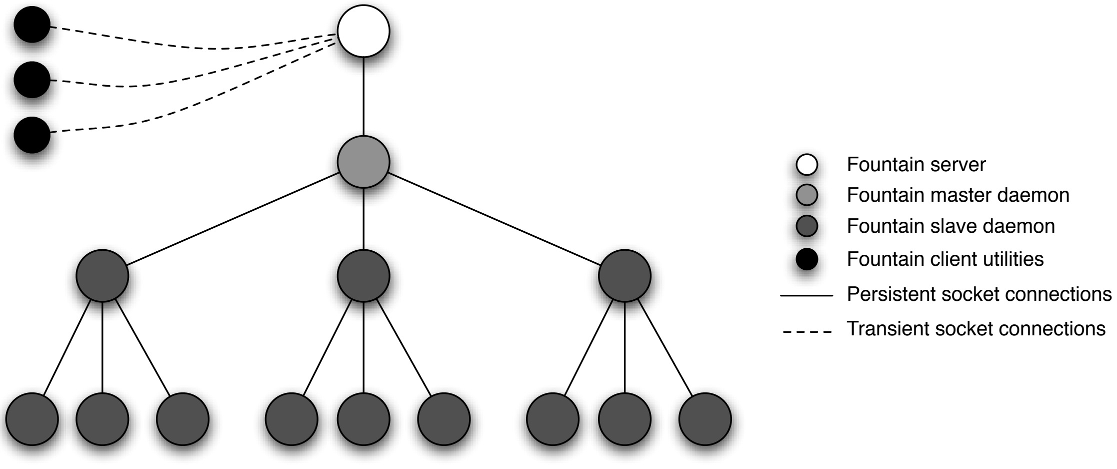

○ Fountain server
● Fountain master daemon
● Fountain slave daemon
● Fountain client utilities
Persistent socket connections
Transient socket connections

Figure 3.1 The four Fountain components.

Each Fountain component is single threaded, written in C++, and uses several common design idioms such as smart pointers, singletons, scope guards [1], assertions [3], and enforcements [4]. These idioms allow for efficient and reusable source code, as well as help in preventing resource leaks such as file descriptors, and memory. Several of these idioms enforce certain axioms, such as only allowing a single instance of a logging object or requiring a wire protocol object to be wrapped in a resource management class. While the functionality of Fountain could have been achieved using more primitive design methodologies, the loss of a stricter semantic interface opens the door for syntactically valid, yet illegal constructs such as multiple logging objects writing to the same log file. For a single threaded application, this is often a moot point since there is only one thread of execution. However, using these powerful idioms paves the way to further extend Fountain into other monitoring domains that may require threading capabilities in the future.

We chose to use a single thread of execution rather than multiple threads to concentrate on efficiency. Applications such as the Apache web server [5] use a thread pool model and place more emphasis on achieving high throughput and quality of service rather than using a minimal amount of resources. At its heart, Fountain is a monitoring application While monitoring perturbation is an unavoidable side effect, we felt using multiple threads of execution would promote higher perturbation. Certain components, such as the master Fountain daemon and the Fountain server may benefit from having multiple threads in the future. However, we have primarily concentrated our development efforts in making them efficient for single threaded purposes thus far.

3.2 Environment and Requirements

Supporting various Linux operating systems has been the primary focus during the development of Fountain due to its general acceptance in the high performance computing industry. However, it should not be overly difficult to port to other UNIX-like operating systems such as IBM's AIX or Apple's Mac OS X due to its use of the GNU autotools build system.

Fountain does not require a global file system or any specialized underlying network hardware. A typical installation will have a front end node of the cluster connected to an external network suitable for users to login, and the compute nodes connected to a separate internal network. While it is not a requirement, the master Fountain daemon and Fountain server typically run on the same node. They communicate over a standard TCP socket and could exist on separate front end or login nodes of the cluster if needed. The Fountain client utilities also do not need to be run on the same node as the Fountain server. Like the master daemon, they also use a TCP socket to communicate with the Fountain server and can connect to it using the service directory.

3.3 Connections

The Transmission Control Protocol (TCP) is a connection-oriented protocol that provides a reliable, end-to-end, and full-duplex byte stream over an unreliable network [29]. The protocol guarantees reliable and in-order delivery of sender to receiver data. To establish a connection between two hosts, TCP uses a three-way handshake to ensure the hosts are fully synchronized before they can send or receive messages. A similar four-way handshake is used to close a connection between the two hosts. While both of these steps ensure a reliable connection, they can result in significant overhead and performance degradation if the messages exchanged between the two hosts is small. To mitigate this overhead, the majority of the connections used by the Fountain components are permanent. Temporary connections are only used for the Fountain client utilities (Figure 3.1) and during the algorithms used to establish and maintain the topology of Fountain daemons, which will be explained in detail in section 4.2.

The use of permanent TCP connections between the Fountain daemons promotes a rigid topology used to exchange messages. This topology minimizes the TCP connection setup overhead and also allows each Fountain daemon to notice and report neighboring node failures as soon as possible. As mentioned in chapter 2, other node monitoring systems utilize different mechanisms to aggregate information into a single system image, such as broadcast messages used by Ganglia, and linear polling used by Supermon. The underlying network structure required by Fountain is fairly minimal. Each node that runs a slave Fountain daemon has to be able to connect to the node running the master Fountain daemon, as well as any other node in the system. The rigid topology used to coordinate the slave Fountain daemons does not indicate nor have any knowledge about the underlying physical network layout.

3.4 Bamboo Library

The Bamboo library provides several common features utilized by all of the Fountain components. The name for the library comes from the Bamboo queue manager, a separate component in the SSS environment. Configuration file management, extensive logging mechanisms, multiple wire protocol implementations, XML message handling [19], and methods to interact with the SSS Service Directory component are just some of the features provided by the Bamboo library.

Each Fountain component utilizes the wire protocol and XML message handling features present in the Bamboo library. The library provides the following wire protocol implementations in order to communicate with other components in the SSS environment:

- **Basic** - a basic wire protocol implementation providing no authentication or encryption, intended solely for transient connections
- **Challenge** - adds a challenge password authentication mechanism to the basic wire protocol
- **Byte count** - adds a header containing the message length to the basic wire protocol making it suitable for persistent connections
- **Challenge byte count** - combines the features of the challenge and byte count protocols into one
- **HTTP** - Implements the HTTP 1.1 protocol specification
- **SSS Resource Management and Accounting password** - Adds encryption and password authentication to the HTTP protocol
- **SSS Resource Management and Accounting symmetric key** - Adds encryption and symmetric key authentication to the HTTP protocol

XML was chosen as the message format for the Bamboo library for multiple reasons. Foremost, it's required by the SSS project that all components communicate using XML messages in compliance with the component schemas. Secondly, an XML message is easily parsed and presents a natural hierarchical structure which is well suited for many types of resource management components.

In addition to the wire protocol methods used to connect to other components, the Bamboo library also provides methods to handle incoming connection requests. A **ServerSocketHandlers** class provides methods to open a listening port using a specified protocol, as well as check for any incoming connection requests and return a pointer to a wire protocol object if one was detected. This method utilizes the `select` system call to multiplex multiple sockets together.

3.5 Summary

Fountain utilizes a component based design that consists of a server, master daemon, slave daemons, and client utilities. Each component communicates with others using XML messages over TCP sockets. The wire protocol and XML message handling facilities are provided by the Bamboo library. Fountain requires no shared file system or specialized physical network layout to accomplish its monitoring goals. The sole requirement is a network of TCP connected hosts where any node can connect to any other node.

CHAPTER 4. FOUNTAIN DAEMONS

There are two types of Fountain daemons. A single master Fountain daemon runs on the head or login node of the cluster, and slave Fountain daemons run on all the cluster's compute nodes. The purpose of each daemon is to collect information from the node they are running on and promptly report neighboring daemon failures. Each slave daemon is arranged in a n-ary tree topology by the master daemon, where each daemon has persistent connections to a parent and any number of children. This topology and the algorithms used to maintain it will be explained in detail in Section 4.2. An example 3-ary topology is shown in Figure 4.1. Both types of Fountain daemons are intended to run without user interaction, hence the word daemon in their name. To aid in development, both daemons support an optional debug mode that enables them to run with a terminal attached, or to run as a daemon and create a log file. For testing purposes on a single computer such as a laptop, the log file uses a unique identifier to allow multiple Fountain daemons to run concurrently without stepping on each other's toes. When compiled for production purposes, the slave daemons do not create a log file, while the master daemon creates a log file in `/var/log/fountain/master`.

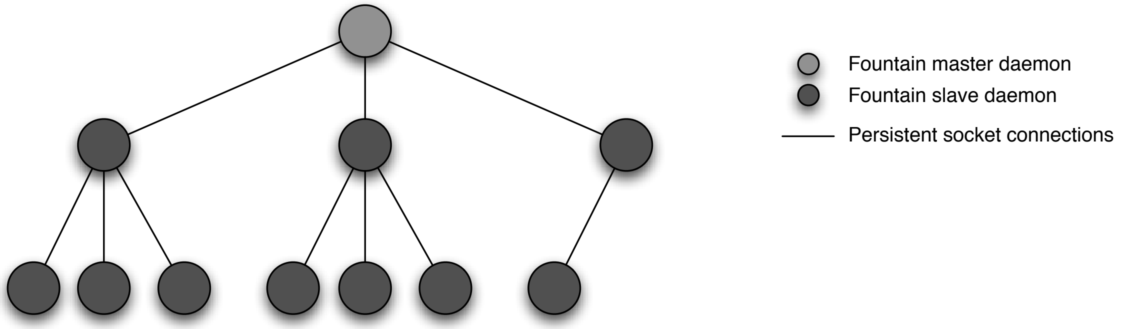


Figure 4.1 Example Fountain tree topology using 3 children per node

As described in chapter 3.1, each Fountain daemon is a single threaded process. To minimize node overhead each Fountain daemon sleeps for up to 500 milliseconds during each execution of its main control loop. The sleep duration comes from executing a `select` system call, during which one of the following three events will happen:

1. An incoming connection request is detected
2. A message on an existing connection is detected
3. Nothing is detected, sleep for 500 milliseconds

The `select` system call provides the semantics to sleep for this duration of time, or return prematurely if conditions 1 or 2 in the above list are met. To manage its connections, a Fountain daemon builds upon the Bamboo Library's `ServerSocketHandlers` class described in section 3.4. By itself, this class is not suitable for the Fountain daemons because it only considers listening sockets created for detecting incoming connections. That is, persistent connections that are maintained as objects on the heap will not have their file descriptors accounted for when performing the `select` system call in the `CheckConnections` method. For optimal performance, a Fountain daemon must be aware of incoming connection requests on its listening socket, as well as messages waiting on existing persistent connections. As an example, each Fountain daemon has a single listening port in addition to connections to its parent daemon and any number of child daemons. If the Bamboo Library's `ServerSocketHandlers` class was used, it would never detect when messages are waiting on these connections during its `select` system call. It would only detect incoming connection requests on the listening port. While this would work since a Fountain daemon will check for messages on all of its connections during its main control loop, it would not be an ideal solution due to the half-second delay between iterations.

To mitigate this problem each Fountain daemon uses a `FountainSocketHandlers` class that inherits from the Bamboo library's `ServerSocketHandlers` class by overloading the `CheckConnections` method. It works in coordination with a Fountain specific wire protocol class to make a distinction between sockets maintained as connections on the heap, and listening sockets used to detect incoming connections. Both types of file descriptors are combined into a single `select` system call to ensure every connection to a Fountain daemon is accounted for.

4.1 Slave Fountain Daemon

A slave Fountain daemon consists of a process running on each compute node of a cluster with two purposes. The first is to collect the static and dynamic monitoring information for the node it is running on, and the second is to report neighboring daemon failures in a timely fashion. In the current implementation, Linux is the only supported architecture because the required monitoring information can be found in the `/proc` file system, and due to its wide acceptance in both the high performance computing industry and research communities. However, 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. In the current implementation, the information collected by each daemon is the amount of configured memory and swap space, amount of available memory and swap space, CPU usage, number of CPUs, CPU architecture, and node operating system. Section 4.1.1 explains how each of these metrics are collected and calculated.

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 daemon, after which it enters its main control loop. Inside this loop, it checks for incoming connections or requests on existing connections as described previously. If there are no incoming connection requests, or data waiting to be read on existing connects, the slave daemons essentially do nothing. They only collect node monitoring information when requested to. That is, the Fountain server pulls data from the Fountain daemons rather than the daemons pushing the data.

The node state information provided by Fountain to the cluster scheduler is not a metric that is collected by a slave Fountain daemon. This information is maintained by the Fountain server, and will be explained in greater detail in chapter 5. The Fountain server assumes a node's state is `Up` if it's present in the Fountain tree topology. Typically, the cluster scheduler depends on an external process manager to actually start the user's job [12] (see figure 1.1). Therefore, the disjoint relationship between Fountain and the process manager may pose somewhat of a problem to the cluster scheduler. Ideally, each Fountain daemon should somehow interface with or detect the process management component to ensure it is working properly and capable of starting user jobs on that node. This could be achieved by each Fountain daemon checking for the pid of the MPD process in the event MPD is the process manager [7]. If the pid is not found, the Fountain daemon should indicate this failure by changing its own node state to `Down` or `Unavailable`. Appendix B describes a prototype process manager called Moso that integrates with Fountain to provide this functionality.

4.1.1 Monitoring Metrics

For our monitoring purposes, there are two types of metrics. Static metrics are those that have a constant value which never changes. An example is the node's number of installed CPUs. While it is certainly possible for a node to add or remove CPUs, we have made the assumption these static metrics never change their values over the lifetime of a Fountain daemon. Dynamic monitoring metrics have values which are expected to change.

The static monitoring metrics include the number of installed processors, their clock speed, the amount of configured memory (RAM), the amount of configured swap space (virtual memory), the node operating system, and the processor architecture. Each of these can be collected from either the `/proc/cpuinfo` or `/proc/meminfo` file. Parsing the output from each of these files is fairly trivial since they are plain text files. However, should their content change in a newer version of the Linux kernel, care should be taken to ensure the correct values are obtained.

The dynamic monitoring metrics include the total CPU utilization, the CPU utilization for each processor, the amount of memory utilized, and the amount of swap space utilized. Obtaining and calculating the dynamic node usage information requires more work than the static information. The amount of available memory can be collected from the `/proc/meminfo` file using the same method as the amount of configured memory, and the percentage of memory utilized can be calculated by dividing the configured metric by the available metric. The CPU usage is calculated based on the concept of a *jiffy*, which is the duration of one tick of the system timer interrupt. This interval has no absolute value since it depends on the host node's clock interrupt frequency. A sample `/proc/stat` file from a multiprocessor system would look like:

```
cpu  589485 135177 168873 2330512064 81131 11441 9053
cpu0 254628 72750 124594 1165251842 39662 5599 4549
cpu1 334856 62427 44278 1165260222 41468 5841 4504
...
processes 127364
procs_running 1
procs_blocked 0
```

The three rows labeled "cpu" are the overall usage, and the usage for the respective processors. The first column is the number of jiffies that the system has been in user mode, the second column is user mode with low priority (nice), the third column is system mode, and the last column is idle task. To calculate CPU usage, equations 4.1, 4.2, and 4.3 are used.

$$jiffies\ busy = \Delta user + \Delta current + \Delta system \tag{4.1}$$

$$jiffies\ total = jiffies\ busy + \Delta idle \tag{4.2}$$

$$cpu\ usage = \frac{jiffies\ busy}{jiffies\ total} \tag{4.3}$$

4.1.2 Reporting Failures

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 tree topology. The connection to a parent node is used to receive and process request messages. An example request is a `query` message, which instructs the daemon to gather its node monitoring information. Slave 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 daemon assumes that neighboring node has failed and it reports the failure to the master daemon. Any other request message received from a child daemon is discarded.

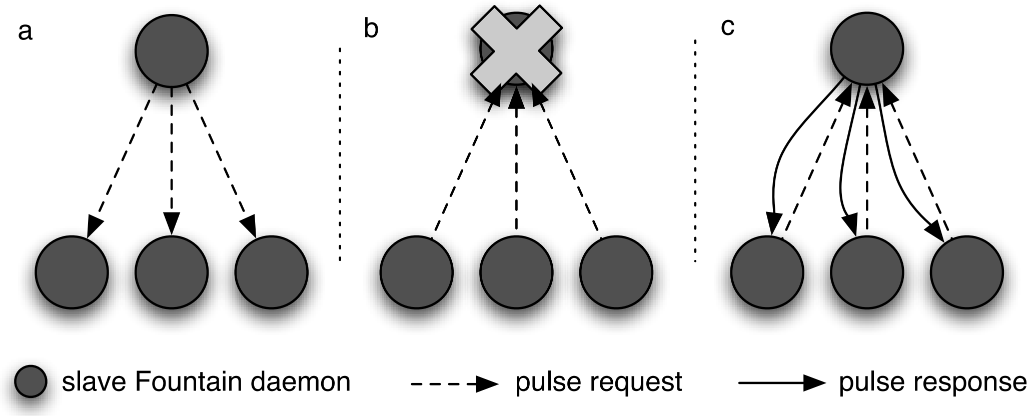


Figure 4.2 The three purposes of the pulse request and response messages.

Pulse messages serve three purposes, they are shown in Figure 4.2. The first purpose, in Figure 4.2a, 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 purpose is shown in Figure 4.2b, 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 4.2c, when the parent daemon has not sent its children a message during a predefined time interval but it's still alive for whatever reason. 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.

4.2 Master Fountain Daemon

A Fountain system consists of a single master Fountain daemon running on the head node of a cluster. The master daemon is exactly the same as the slave daemons, except it has the added requirement of maintaining a topology of slave Fountain daemons. 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. This decision was based on the prior work in [10]. Since this topology will be used to transmit messages containing monitoring information, we anticipate they will grow quite large as the topology of Fountain daemons increases in size. This idea promotes a tree topology in favor of a ring topology since the round-trip-time of messages in a ring scales linearly with the system size. In our design the degree of the tree is configurable, Chapter 6 will present some conclusions about the optimal topology degree based on node query performance results.

The Fountain tree topology is designed as a complete n-ary topology, where each entry has at most `n` children. All levels of the tree are full except for the bottom level, which is filled from left to right. Internally the topology is maintained as an array, Figure **??** shows a sample binary tree topology with 7 nodes and the parent/child connections. Equations 4.4 and 4.5 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 array. Note that `a` can have values between `1` and `n` inclusive.

$$parent = \lfloor (b-1)/n \rfloor \tag{4.4}$$

$$child = n * b + a \tag{4.5}$$

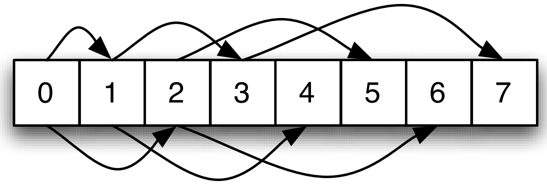


Figure 4.3 Binary tree topology represented as an array.

Like the slave Fountain daemons, the master Fountain daemon opens a listening socket during its initialization process, where it also initializes the tree topology to contain itself as a tree with a single node. To add slave daemons to the topology, they connect to the master daemon and initiate the tree establishment algorithm. To maintain the tree topology, Fountain has four states: `idle`, `join`, `recovery`, and `rebuild`. The tree is by default in the idle state, and can transition to any of the other states by the conditions shown in Figure 4.3. The other three states are used to maintain the tree topology in the event of node failures. They will be described in detail in the following sections.

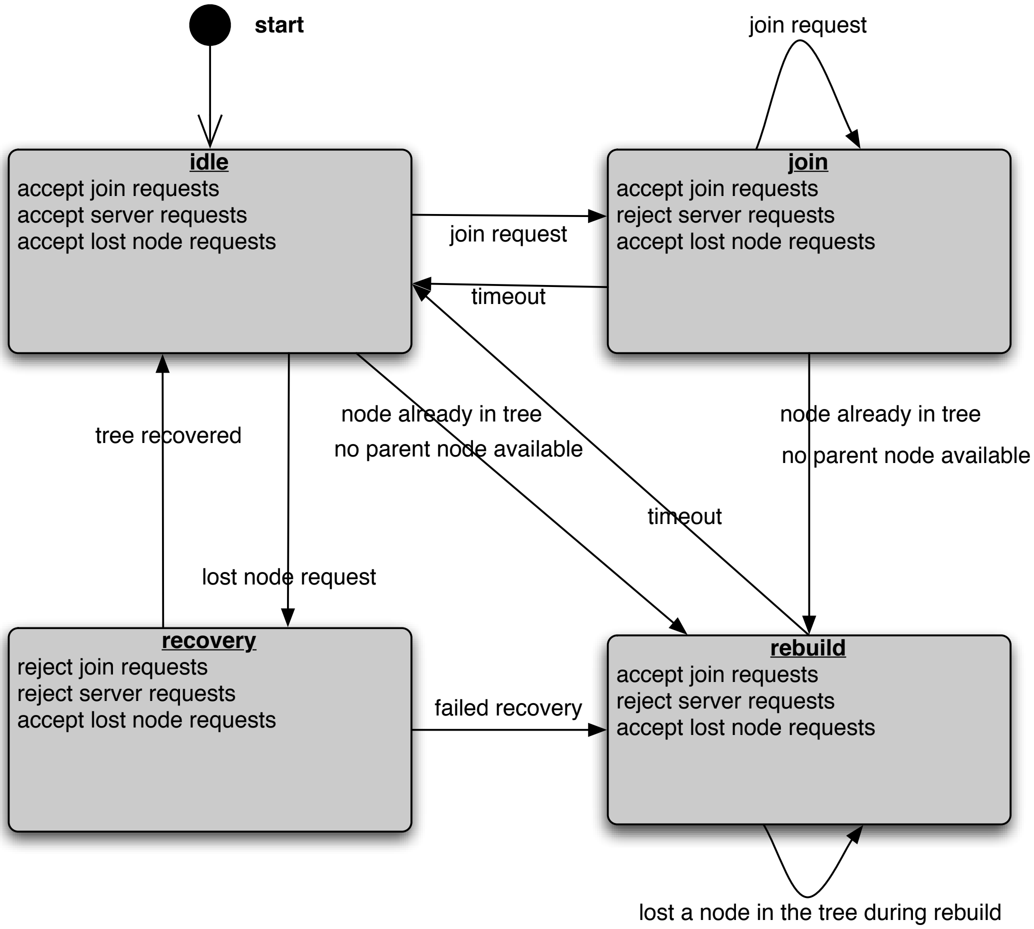


Figure 4.4 Tree topology state transitions

4.2.1 Tree Establishment Algorithm

When a slave Fountain daemon first connects to the master daemon, they initiate the tree establishment algorithm. This algorithm uses a three way handshake to introduce new slave Fountain daemons the tree topology. After receiving the initial connection request, the master daemon looks up the available parent 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. If 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 by one. If the connection is not successful, the slave daemon responds to the master daemon with a failure response and the tree topology data structure is not updated. 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 4.4.

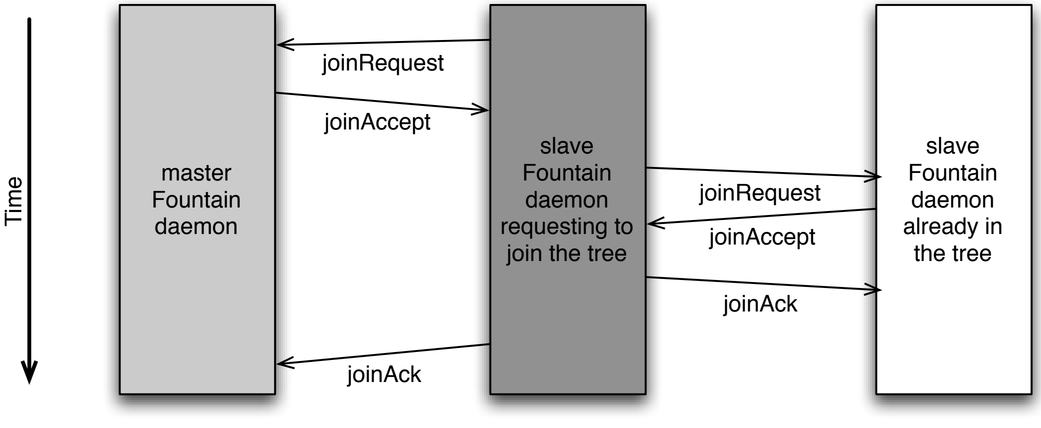

Time
master Fountain daemon
joinRequest
joinAccept
slave Fountain daemon requesting to join the tree
joinRequest
joinAccept
joinAck
slave Fountain daemon already in the tree
joinAck

Figure 4.5 Tree establishment message sequence

4.2.2 Tree Recovery Algorithm

In addition to the tree establishment algorithm, the master Fountain daemon uses two algorithms to maintain 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 or to close their connection unexpectedly. It could constitute a kernel panic by the node operating system, a power loss event, failed ethernet switch, or any number of other causes. 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 [10] and [9]. When a slave daemon in the tree topology fails, its parent daemon and all child daemons notice the failure when their connection is closed unexpectedly, or closed in response to a failed pulse request. They will then attempt to report this failure to the master daemon. The reason all neighbors of the failed node have to report the failure is because each slave daemon is only aware of its direct neighbors. Should its parent fail, it has no knowledge of other daemons in the system, and therefore cannot connect to a new parent without help from the master daemon. Additionally, it is possible for two or more adjacent daemons in the topology to fail concurrently. In those events, we cannot rely on a single daemon to report the failure. For example, if the recovery algorithm relied on the failed node's parent to report its failure, not all failed nodes would be reported in the event two neighboring nodes fail concurrently. In Figure 4.5, if nodes 1 and 4 both fail at the same time, and nodes 13, 14, and 15 rely on node 1 to report the failure of node 4, node 4's failure will never be reported and its children will not be connected to the Fountain tree topology.

Upon receiving a lost node 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 node requests. That is, when the Fountain server sends any request 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

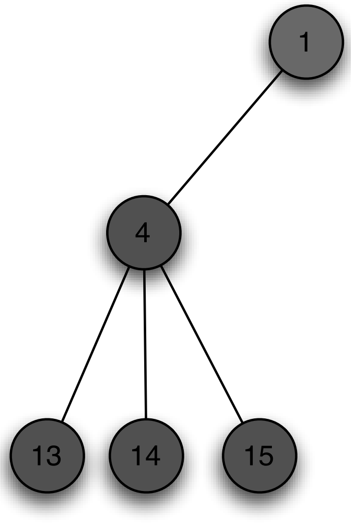


Figure 4.6 A segment of a 3-ary tree topology

are also rejected by the master 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, where a neighboring daemon is either the lost daemon's parent or one of its children. After all of the failed daemon's neighbors have contacted the master daemon to report the failure, the master daemon selects a replacement daemon from the tree topology. To minimize the number of slave daemons affected by this recovery algorithm, the replacement daemon is always the last daemon to join the tree topology because it only has a single connection to its parent daemon. After the replacement daemon successfully joins the failed daemon's parent, the master daemon informs the failed daemon's children to join the replacement daemon. The master daemon then sets the tree state back to idle so requests from the Fountain server can be processed.

The tree recovery algorithm is shown in Figure 4.6 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. This shows how recovering a 3-ary tree topology only affects a maximum 5 out of 16 nodes.

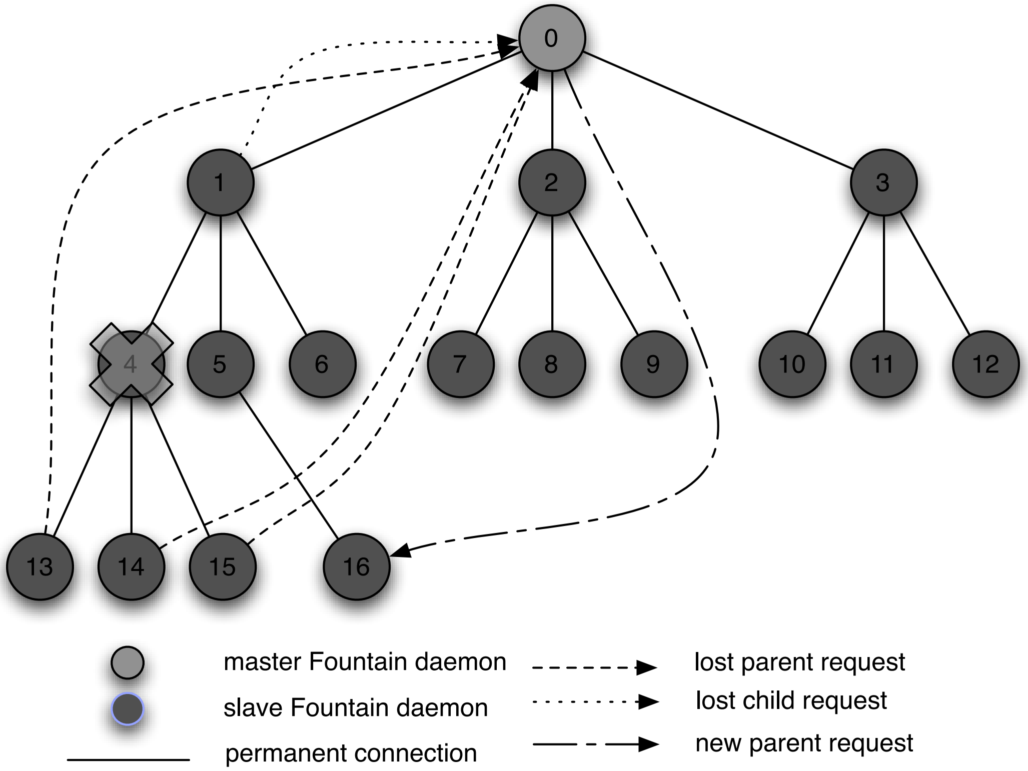


Figure 4.7 Fountain tree topology recovering from a failure of node 4, which will be replaced by node 16

4.2.3 Tree Rebuilding Algorithm

If multiple nodes in the cluster fail at or near the same time, the master Fountain daemon will accept the first lost node request and reject all subsequent requests for different lost nodes by informing the daemon that reported the failure to try again later. This could cause a race condition if a single slave daemon has both a child and grandchild daemon fail concurrently. The race condition arises because the failed child daemon may or may not have reported the failure of the grandchild daemon before its own failure. This race condition is mitigated by starting 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 connections. This will cause its children to notice the closed connection and report a lost node. The master daemon in turn will respond to these lost node requests by telling the slave Fountain daemons 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.

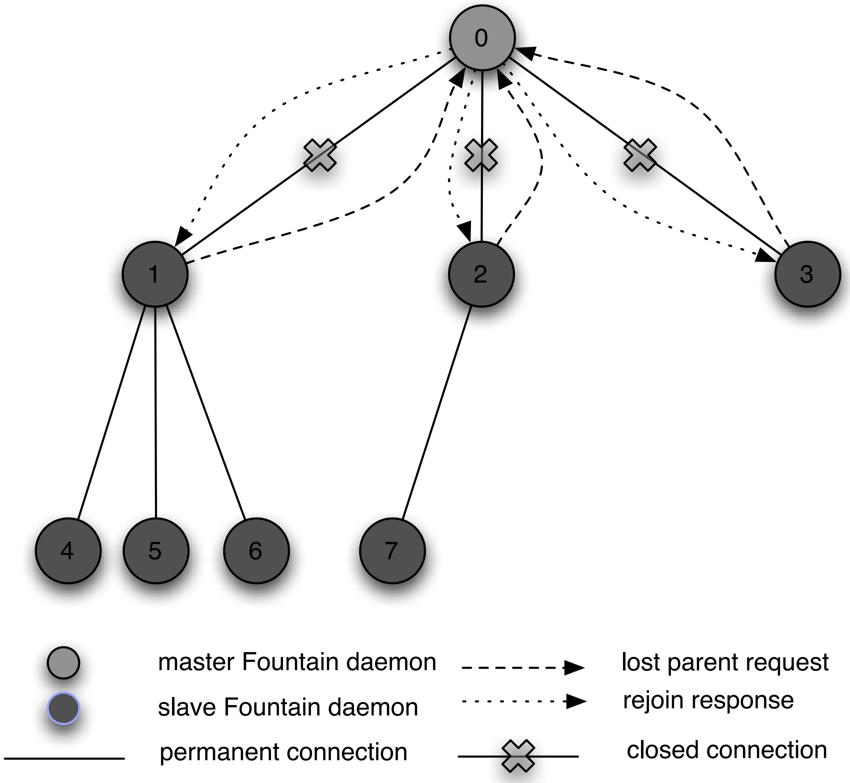


Figure 4.8 The first step in rebuilding a Fountain tree topology of 7 slave daemons

The first step in the tree rebuilding process is shown in Figure 4.7. The second step is shown in Figure 4.8, there is no third step in this example since the tree topology only consists of 7 slave Fountain daemons, which is two levels deep for a 3-ary tree. Note that both of these figures do not show how the tree topology enters the rebuild state. In practice, multiple neighboring daemons would have to fail concurrently to enter the rebuild state as described previously. They also do not show each slave daemon rejoining the tree topology, which happens after they close their parent and child connections.

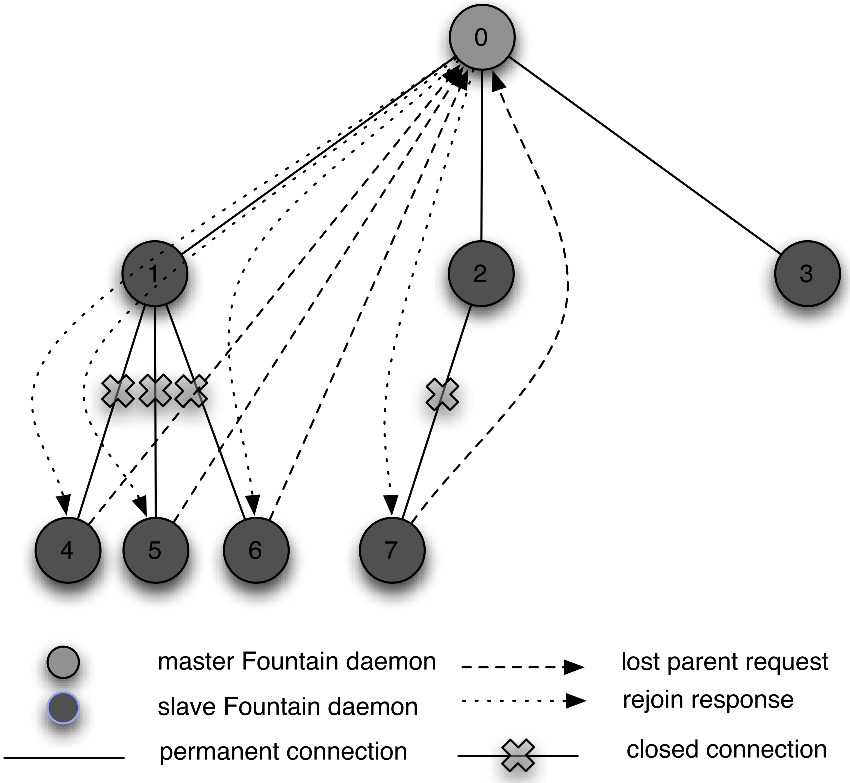


Figure 4.9 The second step in rebuilding a Fountain tree topology of 7 slave daemons

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. When it happens, it will be detected by the master daemon's direct children. Since each slave daemon has no knowledge if its parent is the master daemon or another slave daemon, they attempt to connect to the master daemon and report its own failure. Obviously this will not succeeded since the master daemon has failed and cannot accept connections if it does not exist. 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.

Future work exists to investigate other methods for recovering from a master daemon failure. The current method to recover from such a failure requires a system administrator to either reboot the node, or restart the master Fountain daemon. If the cluster has multiple login nodes or service nodes, it may make sense to run a backup master daemon similar to the backup SLURM control daemon [13]. In this fashion, the slave Fountain daemons could fall back to using one of the backup master daemons if the main one has failed. Care would have to be taken to ensure the master daemon and all of its backups maintain a consistent state. A checkpoint file could also be used by the master daemon to periodically write the tree topology structure to disk, then look for such a file during its initialization step when its rebooted after a failure.

4.2.4 Verification of Tree Topology Algorithms

Appendix A describes the process of using the model checker SPIN to verify the tree establishment algorithm used by the master Fountain daemon. Verification of this algorithm is important because both the tree recovery and rebuilding algorithms depend on an accurate topology to perform correctly.

4.3 Summary

There are two types of Fountain daemons, slave daemons that run on each compute node of a cluster and a single master daemon that runs on the head node. Each slave Fountain daemon is responsible for collecting node information from the `/proc` file system. The master Fountain daemon arranges the slave daemons in a logical tree topology to promote scalability for larger cluster sizes. To recover from individual and multiple node failures, the master daemon utilizes three algorithms to maintain the tree topology.

CHAPTER 5. FOUNTAIN SERVER

The Fountain server is responsible for presenting a single system image of the cluster to other components in the SSS environment in accordance with the SSS Node Object specification. This chapter will outline the interface the Fountain server uses to present the node information to other components, explain the design choices used when creating this component, and describe how it interacts with the Fountain daemons. Two extensions for the Fountain server to extend its monitoring capabilities beyond node specific sources will also be presented.

5.1 Interface

The Fountain server acts as the gateway between the Fountain daemons and other SSS components. With that in mind, it's interface should be both easy to understand and to use. During it's initialization process, the server opens a listening port to accept incoming connections. After opening this port, it registers itself with the system wide Service Directory component so other components are aware of what port and protocol to use when connecting to the Fountain server. After some additional initialization work, the server enters its main loop where is waits for incoming connection requests.

Fountain uses the SSS Resource Management and Accounting (SSSRMAP) protocol to communicate with other components. This protocol defines a connection oriented, XML based, application layer client-server protocol for resource management and accounting software components developed as part of the Scalable Systems Software Center [25]. The SSSRMAP protocol itself only defines a framing protocol that includes security elements. It uses the XML schema shown in Appendix C.

When it detects an incoming connection, the Fountain server expects to receive a message formatted using the SSS Node Object specification. This specification is a detailed XML syntax used for representing all the static and dynamic information for a single compute node in a cluster. A simple node monitor request is shown in Figure 5.1, and a more detailed request is shown in Figure 5.3. The responses to these requests are shown in Figures 5.2 and 5.4. The interface Fountain presents is very flexible with respect to the data that can be returned by combining multiple `Get` and `Where` elements together. For example, the cluster scheduler could query Fountain for all the nodes that match a parallel job's requirements. It's also very useful for a system administrator to quickly query Fountain for all the nodes with a state that is not `Up`, indicating some sort of failure.

For metrics that support units, such as available memory, the Fountain server supports an optional `units` attribute for each `Get` element. The units may take one of the following values: Kilobytes (`KB`), Megabytes (`MB`), Gigabytes (`GB`), Terabytes (`TB`), or Petabytes (`PB`). Internally, each value is stored in kilobytes; so the conversion to the requested units is a trivial shift instruction. Figures 5.3 and 5.4 show an example of the `units` attribute.

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

Figure 5.1 Simple node monitor query request

The Fountain server performs strict syntax checking on all requests it receives to ensure they are formatted in accordance with both the SSSRMAP specification and the SSS Node Object specification. The name attribute specified in the `Get` and `Where` elements must be supported by Fountain, otherwise it will respond with an error. For example, requesting a `Get`

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

Figure 5.2 Simple node monitor query response

element with a `name` parameter of `NodeState` is supported, however a value of `NodeColor` is not supported. In the spirit of completeness, Fountain supports an optional `required` attribute with a value of `true` or `false` for each `Get` element. While this feature is not described in the SSS Node Object specification, if its value is `true`, it enables consumers of the data returned by Fountain to handle the error checking of the response message.

5.2 Design

To aggregate the information of every node in a cluster into a single system image, the Fountain server utilizes a node monitor data structure. The information returned in a `query` response is collected from this data structure. This structure itself is a `map` container from the C++ Standard Template Library. Each entry in the container has a key and value pair, where

```
<Envelope>
  <Body actor="root">
    <Request action="Query">
      <Object>Node</Object>
      <Get name="NodeId"></Get>
      <Get name="Arch"></Get>
      <Get name="OpSys"></Get>
      <Get name="State"></Get>
      <Where name="State" op="eq">Up</Where>
      <Get name="Configured/Processors"></Get>
      <Where name="Configured/Processors" op="ge">2</Where>
      <Get name="Available/Memory" units="MB"></Get>
      <Where name="Available/Memory" op="ge" units="MB">128</Where>
    </Request>
  </Body>
</Envelope>
```

Figure 5.3 Extended node monitor query request

each key must be unique. The key for each entry is a unique identifier for each node in the cluster provided by the Fountain daemon running on that node. In our current implementation, this identifier is the hostname truncated to the first period. The value for each entry is `NodeData` object capable of storing various node statistics, such as CPU usage, memory usage, swap usage, etc. The actual object type used depends on configuration parameters, so it's chosen at compile time. This policy based design will be explained in section 5.3.

Initially, the node monitor data structre is empty and can only be erased through the Fountain administrative utility. It can be populated by each of the following conditions:

1. Parsing a query response from the master Fountain daemon

2. Parsing the node list file during the Fountain server's initialization

3. Discovering a *server specific* data source

Each of these conditions will be explained in the following sections.

```
<Envelope>
  <Body actor="root">
    <Response action="Query">
      <Count>1</Count>
      <Total>37</Total>
      <Data name="NodeList" type="xml">
        <Node>
          <State>Up</State>
          <NodeId>m17</NodeId>
          <Arch>ppc</Arch>
          <OpSys>Linux</OpSys>
          <Configured>
            <Processors>2</Processors>
          </Configured>
          <Available>
            <Memory units="MB">819.5</Memory>
          </Available>
        </Node>
      </Data>
      <Status>
        <Value>Success</Value>
        <Code>000</Code>
        <Message>1 node(s) found</Message>
      </Status>
    </Response>
  </Body>
</Envelope>
```

Figure 5.4 Extended node monitor response

5.2.1 Querying the Fountain daemons

The first condition to populate the node monitor data structure comes from querying the Fountain daemons. A query in this sense is different than the query request the Fountain server expects to receive from clients such as the cluster scheduler. To accomplish this, the Fountain server has a persistent connection to the master Fountain daemon (Figure 3.1). Typically, both components run on the head node of the cluster. This is not a requirement however, the connection between Fountain server and master Fountain daemon is a TCP socket just like the ones used to connect slave Fountain daemons together. This connection is used by the Fountain server to send requests and receive responses. The `query` request is sent at an interval specified in the Fountain server configuration file. After the master Fountain daemon receives a `query` request, it processes it, forwards the request to all the slave Fountain daemons and waits for their responses. After collecting all of the query responses, the master Fountain daemon sends a query response message to the Fountain server. This process is shown in Figure 5.5. The server then processes this query and inserts any new entries into the node monitor data structure.

As described in Chapter 4, each Fountain daemon collects both static and dynamic information about the node it is running on. The static information is only collected at initialization and the dynamic information is collected in response to each query request. The Fountain server has two distinct query request types, `Configured` and `Available`. As implied by their names, A `Configured` query request asks each Fountain daemon to report both their static and dynamic node information, while an `Available` request only asks for the dynamic information. Before sending the Fountain daemons a query request, the Fountain server checks if it encountered an entry in the previous query response that was not in the node monitor database. If so, then a `Configured` query request is sent and the new node's information will be added. Otherwise, a `Available` query request is sent. These two query request types have the disadvantage of a slight delay when introducing new Fountain daemons into the system since it takes two query requests before they are added to the node monitor data structure. However, doing so prevents the static node monitoring metrics from being included in each query response, thereby conserving network bandwidth.

The second condition to populate the node monitor data structure occurs during the Fountain server's initialization process, where it reads a file containing a list of nodes in the cluster. The nodelist file is optional, but it's useful to create an entry in the node monitor database for a node that may otherwise not be found. For example, if a node is physically disconnected from the cluster for whatever reason, it will never respond to a query request from the Fountain server because no Fountain daemon can run on that node. Therefore, its entry would never be created in the node monitor data structure. This condition is also useful to artificially populate

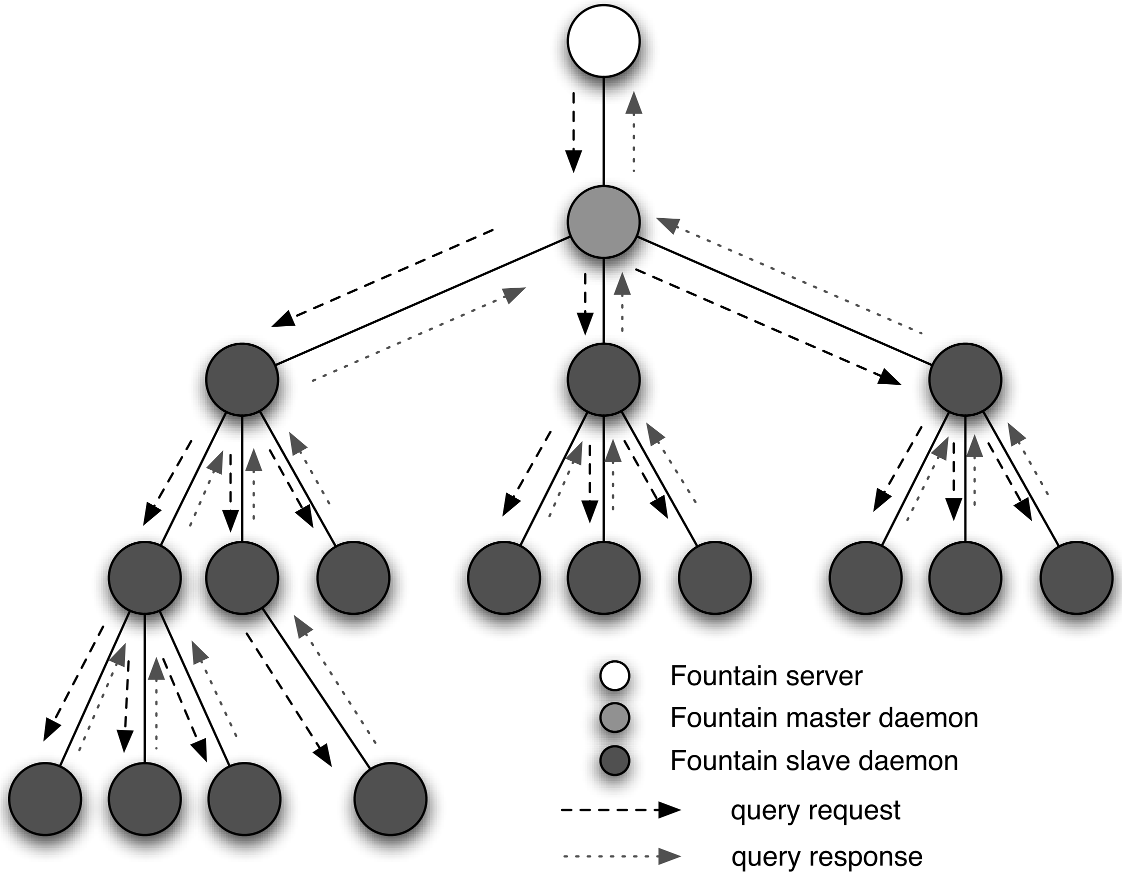


Figure 5.5 Querying the Fountain daemons for node status information.

the node monitor data structure with a large number of entries for testing purposes.

We use the term data structure and database interchangeably to describe the node monitor data structure object. It is not a traditional database like MySQL, Microsoft Access, or IBM's DB2. Instead, it stores its information entirely in memory and uses no backing store in the event the Fountain server fails. A future research question exists to determine if an external database should be used to manage the node statistics. For larger cluster sizes, the node monitor database memory usage may become higher than desired. Utilizing an external database may prove to provide both performance benefits as well as lower memory usage by the Fountain server.

Obtaining accurate node state information is a high priority since the primary consumer of this data is the cluster scheduler. This information is maintained by the Fountain server for each entry in its node monitor database. For node monitoring purposes, Fountain has four values for a node's state:

1. `Invalid` - default state
2. `Up` - ready to run user jobs
3. `Down` - cannot run user jobs
4. `Unavailable` - administratively offline

Each state and its available transitions are shown in Figure 5.6. When creating an entry in the node monitor database, its state is set to `Invalid` by default. After processing a query response from the Fountain daemons, each entry that had information updated has its state set to `Up`. However, before processing a query response, each entry in the node monitor database has its state set to `Down` if it was previously `Up`. If this was not done, then nodes in the Fountain tree topology that are lost will continue to have a node state value of `Up` even though their status is unknown.

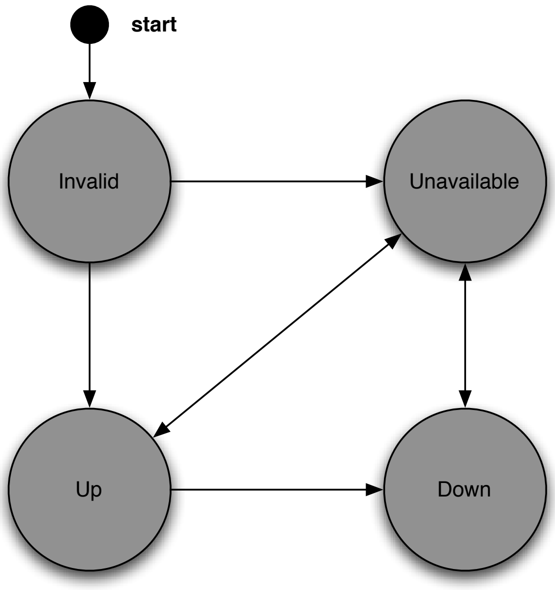


Figure 5.6 Node state transitions.

There exists a slight delay between when a node is actually down and cannot run user jobs, and when the Fountain server sets that node's state to `Down`. This delay is due to the query interval. A Fountain daemon may respond to a query request from the Fountain server, then die immediately afterwards for whatever reason. If the cluster scheduler attempts to schedule a job to run on the down node it may not succeed because Fountain thinks the node's state value is `Up` rather than `Down`. It's state will not be changed to `Down` until the next query interval when that Fountain daemon fails to respond to the query request. This delay could be mitigated by sending a message from the master Fountain daemon to the Fountain server whenever a node in the tree topology is lost. Our current implementation of Fountain has not implemented this feature as of yet.

As discussed in Section 4.1, it may be advantageous to allow the Fountain daemons to report their state. These values could indicate online but not accepting user jobs state, an administrative offline state, or anything else depending on the health or status of the process management daemon. In our current implementation we do not support this feature because the cluster scheduler used in the SSS environment expects node state values of either `Up`, `Down`, or `Unavailable`. We do support a method to allow a system administrator to set a node's state to `Unavailable`, which otherwise is a state never achieved by a node itself. This state is sticky, it will only transition to other states if requested to do so by the system administrator. This state is useful because it indicates the node is currently unavailable to run user jobs, but it is alive and well. The interface to set node states is achieved through the fountain administration utility explained in section 5.4.2.

5.3 Extensibility

The Fountain server was designed as a system capable of monitoring a wide variety of data sources. Thus far, the only data source that has been discussed is the information collected by the Fountain daemons. This information is specific to the node running the Fountain daemon described in chapter 4. Nodes in a cluster often contain other resources beyond the metrics monitored by the Fountain daemons. Multiple interconnects such as gigabit ethernet, InfiniBand, or Myrinet, and parallel file systems such as PVFS, Lustre, or GPFS are all common in both medium and large size clusters. In some cases, such information may be desired by other components in the SSS environment or by a system administrator. We call these data sources *server specific* data sources because their status does not need to be monitored by a Fountain daemon, and can instead be monitored by the Fountain server directly. Often such systems provide an external mechanism for assessing their status or collecting performance metrics, so integration into the Fountain server is fairly trivial.

5.3.1 Node Monitor Database

The node monitor database described earlier utilizes a policy based design making it both highly extensible and easily adaptable to new data sources. In brief, policy based class design enables the creation of a class with complex behavior by combining smaller classes together [2]. There are multiple ways to achieve such a design, such as a large unwieldy interface, the use of multiple inheritance, or templates. We chose to use the template solution because they generate code at compile time based on types provided by the user. This idea relates extremely well to our node monitor database idea, where the objects stored depend on specific arguments selected by the user during Fountain's configuration script.

Each policy class establishes an interface pertaining to a specific aspect of the problem. The "hello world" of policy based class design is a smart pointer class, where there are policies to manage dereferencing, reference counting, threading, and deletion. Such a smart pointer class presents a very rich extensible interface where each policy class manages an orthogonal piece of the interface to the class. Our implementation of a node monitor database policy design has a `NodeData` host class that supports two policy classes. A `statistics` policy to maintain statistics for each node in the cluster, and a `network` policy to maintain network information. Each policy requires the following interface:

- A protected explicit constructor taking a `string` parameter
- A protected `bool supportGetElement()` method taking a single `string` parameter that returns true or false if the object supports the named `Get` element

- A protected `bool satisfiesWhereElement()` method that returns true if the named `Where` element meets the criteria

- A protected `void appendInfo()` method that appends the object's requested information to the named XML element

These two policy classes are required to be orthogonal since both the node statistics and network policy can co-exist with one another. An example of two orthogonal classes implementing the statistics policy and network policy would be the Fountain daemon class for maintaining information provided by a Fountain daemon, and the InfiniBand network class for maintaining InfiniBand network information. Non-orthogonal `NodeData` policies consist of two or more policies that monitor the same type of data source, such as an ethernet network policy and an InfiniBand network policy. While it's possible that certain clusters may have multiple connectivity options between its compute nodes, we have only considered monitoring a single network data source at this time. Expanding this concept to monitor a second or third network would require adding and implementing additional policy classes to the `NodeData` interface.

By using a policy based design, the node monitor database requires its policies to be defined at compile time since the type of a template parameter has to be statically typed. Both the `statistics` and `network` policy classes for the node monitor database are chosen from typedefs exposed by the Fountain server's `ServerData` object.

5.3.2 Server Data Sources

The Fountain server gathers the server specific information by instantiating a `ServerData` object during its initialization process. This `ServerData` class prescribes a single `DataSource` policy with the following interface:

- A public `NodeDataType` typedef describing the `NodeData` object to use for this data source

- A public `HashTable` typedef describing the container type to use for the Fountain server's node monitor database

```
namespace InfiniBand {
  class Network { /* methods for discovering InfiniBand go here */ };
  class DataSource {
  public:
    typedef NodeData<FountainNodeStatistics, Network> NodeDataType;
    typedef std::map<std::string, NodeDataType> HashTable;
    void discover();
    void update();
    void appendInfo();
    void parseNodelistFile();
  };
}
```

Figure 5.7 Sample implementation of the DataSource policy

- Four methods: `discover()`, `update()`, `appendInfo()`, and `parseNodelistFile()`

The Fountain server uses the `HashTable` and `NodeDataType` typedefs to instantiate its node monitor database. An example class implementing this policy for monitoring an InfiniBand network is shown in figure 5.7. We designed server specific data sources with the idea of two separate events to monitor their resources. A discovery event discovers the resource and collects any pertinent information, which is called during the construction of a `ServerData` object when the Fountain server initializes itself. Optionally, it can be forced to rediscover its resources on demand by using the Fountain administration utility. An update event updates the monitored information for the data source at a regular interval, similar to how the Fountain server queries the Fountain daemons. The appendInfo method is called when Fountain is building a message in response to a node monitor query request.

In our current implementation of Fountain, we only support a single server data source. The type of `ServerData` object instantiated by Fountain is chosen from the configure script [8] with the help of a compile-time if/else statement:

```
template <bool flag, typename T, typename U>
struct Select {
  typedef T  Result;
};
```

```
template <typename T, typename U>
struct Select<false, T, U> {
  typedef U Result;
};
```

This idiom can be used by specifying a boolean constant as the first parameter, then two types to chose from like so:

`Select<false, int, double>::Result`

Where the result would evaluate to `double` in this case. For our purposes of choosing the `ServerData` type, we can use the constant values in the `config.h` header generated by the configure script. It is our assumption that the configure script will generate the correct boolean values in order for the proper node statistics and network policies types based on the user's arguments.

As mentioned when we described the `NodeData` host class, there are multiple ways to add support for additional server data sources in the future. The easiest method would be to run multiple Fountain servers where each one is configured to monitor a different data source. Another method using a single Fountain server would be to instantiate multiple `ServerData` objects during the Fountain server's initialization step, then discover and update each one separately. Alternativly, a second `DataSource` policy class could be used for the `ServerData` object if the two data sources are more closely coupled and require coordination to adequately monitor their resources.

5.3.3 InfiniBand Network Monitoring

Our first component to implement the Fountain server's `DataSource` policy is a module capable of monitoring InfiniBand network performance. InfiniBand is a modern interconnection architecture that utilizes a bidirectional serial bus to achieve high bandwidth and low latency capability. It defines a system area network for interconnecting nodes, where each node has a Host Channel Adapter (HCA) that maintains both send and receive queues for communicating with other nodes. Features such as Remote Direct Memory Access (RDMA) allow one sided operations to move data directly into the memory of a peer node without requiring a receive operation to be posted before a corresponding send. InfiniBand is becoming a popular interconnect choice for large scale clusters due to its high performance and relatively moderate price point per node.

Monitoring a cluster's network resources can be beneficial in two areas. First, a system administrator could detect and alleviate a bad network adapter by visualizing certain error counters presented by the InfiniBand network monitoring software. Secondly, a user running a parallel job could benefit from visualizing potential hot spots in the network as his or her job is executing. This could allow the user to modify the MPI collective operations for their parallel job. Our desire to extend Fountain's monitoring capabilities into this domain came from the first ever floor-wide InfiniBand network at the 2005 Supercomputing conference in Seattle, WA [26]. This network connected industry vendors, research laboratories, the keynote stage, and remote sites together to achieve a high performance InfiniBand network. The unique design of the network required monitoring tools to quickly locate and fix network problems. While our InfiniBand module for Fountain was by no means complete, it gave us much needed insight to add improvements to effectively monitor such a large heterogeneous network.

5.3.3.1 Open InfiniBand Alliance

To monitor a cluster's InfiniBand network, we chose to use the software stack developed by the Open InfiniBand Alliance (OpenIB) [21]. The OpenIB group provides a high performance open source InfiniBand application programming interface that supports HCAs from multiple vendors, and has been accepted into the Linux kernel. The OpenIB software includes various layers above the InfiniBand hardware drivers, in both kernel space and user space, shown in Figure 5.8. With their software distribution, the OpenIB group provides various management utilities to monitor the status of an InfiniBand network. Our InfiniBand module for Fountain is primarily based on the `ibnetdiscover` utility to discover each HCA and switch in the network, and the `perfquery` utility to query a specific node for port counter information.

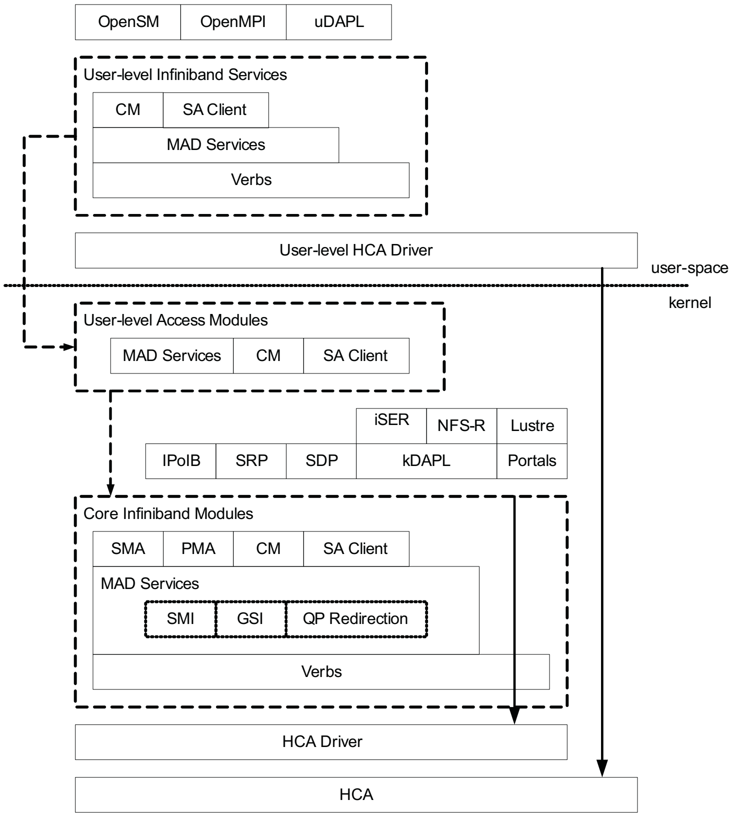


Figure 5.8 OpenIB architectural components.

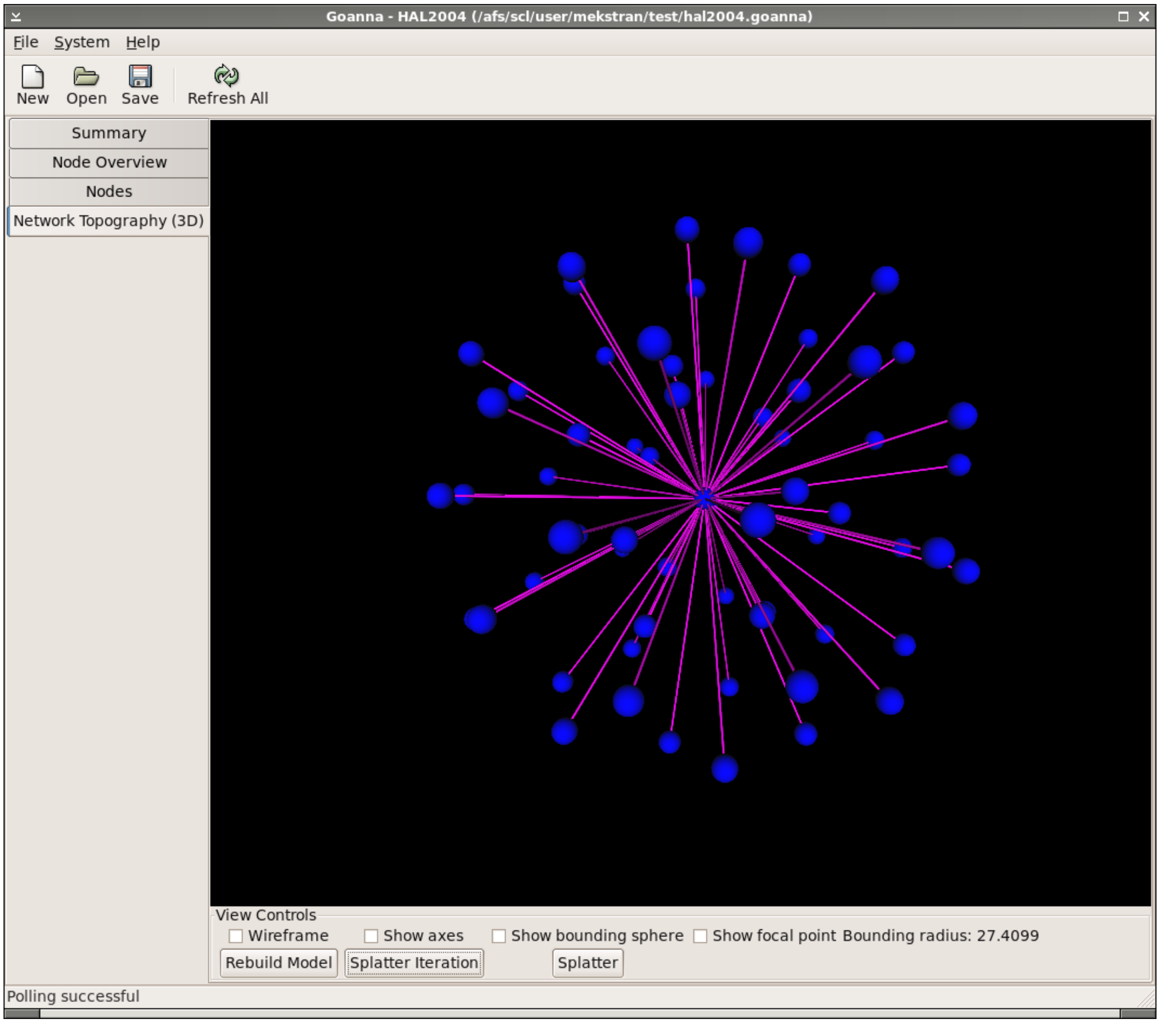
5.3.3.2 Integrating OpenIB into Fountain

The Fountain InfiniBand component provides an `InfiniBand` namespace to encapsulate all its data structures and data types in order to prevent naming conflicts with the rest of the Fountain code base. Inside this namespace, the `DataSource` class implements the Fountain `ServerData` policy. Two other classes inside the InfiniBand namespace are used to keep track of network information. The `Node` class maintains information about each node in the network. This name may be somewhat misleading, since a `Node` object can represent both a compute node in the cluster, as well as an InfiniBand switch. Internally, each `Node` object makes this distinction since it's an important part of network discovery. The `Port` class maintains information about each port of a `Node` object. This information consists of send and receive throughput, error counters, and the Globally Unique Identification Descriptor (GUID) and port number of the Node object its remotely connected to.

The InfiniBand network discovery method originates its discovery process from the node running Fountain server. We make the assumption that HCAs are only connected to switches and not other HCAs, while switches can be connected to any node type. To maintain a list of InfiniBand nodes discovered, the discovery method uses a map container from the C++ Standard Template Library. Each entry in the map uses the InfiniBand node's GUID as the key, and a Node object as the value. If a switch is discovered, every port of the switch is recursively discovered. To prevent loopbacks, GUIDs are compared to ensure uniqueness before discovering each switch. A maximum number of hops away from the originating discovery node is enforced to prevent excessively long network discovery times.

Once a node discovered in the network, the Node object uses the OpenIB umad interface to perform an smp_query request for the IB_ATTR_NODE_INFO information for that node. This information will contain the node's GUID, its type (either Switch or HCA), its number of ports, and some additional information that is not used by Fountain. For each newly discovered node who's GUID does not already exist in the container of previously discovered nodes, the Node object is added to the container. Each Node object also maintains a map container of Port objects to represent each port the HCA or Switch has. Each entry in this container of ports indicates a valid port that is connected to another InfiniBand node, either a switch or an HCA. For each port, certain counters are collected using the umad interface. These counters provide the number of packets sent and received, the number of bytes sent and received, and the symbol errors.

The update method is called periodically during the Fountain server's main loop. It iterates through each node that was discovered during the network discovery step and sends an smp_query request to collect the port counter information for each port the node has active. This information will be stored in the node's Port object, which maintains a timer between updates so send and receive throughput can be calculated for each port. This information can be used to construct a visual representation of the network and present detailed bandwidth usage information to a systems administrator, such as the prototype Goanna application shown in Figure 5.9.



Goanna - HAL2004 (/afs/scl/user/mekstran/test/hal2004.goanna)

View Controls

□ Wireframe □ Show axes □ Show bounding sphere □ Show focal point Bounding radius: 27.4099

Polling successful

Figure 5.9 Goanna InfiniBand network visualization.

5.3.3.3 Node Monitor Integration

Thus far we have shown how Fountain can discover and monitor InfiniBand network resources. However, Fountain needs to make this data available to clients in accordance with its interface presented in section 5.1. Since the InfiniBand network discovery method has no knowledge of the Fountain node daemons, two methods are provided to coordinate a mapping between the InfiniBand nodes discovered and their corresponding Fountain daemons. The first method relies on each Fountain daemon to discover its InfiniBand adapter's GUID during its regular monitoring step. This information is propagated to the Fountain server when the Fountain daemon responds to a query request.

Alternatively, Fountain can be configured without support for the Fountain node daemons and with support for InfiniBand network monitoring. In this case, the Fountain server can run on a single node attached to an InfiniBand network and present this information to an external application for processing or visualization. The discovery method will populate the node monitor database with an InfiniBand node (either HCA or switch) found during the discovery process. Instead of a regular hostname identifier provided by the Fountain daemon for that node, the node's GUID will be used as the key in the node monitor database. Optionally, a `nodelist` file can be read by Fountain when it's first launched or as requested by the user via a `SIGHUP` signal. The contents of this file contain a hostname to GUID mapping on each line, which is useful for a small number of nodes where a textual identifier may be more appealing than the hexadecimal GUID.

A sample `Node` element in a query response is shown in Appendix D. In this example the Fountain server was configured without support for the Fountain node daemons, so the NodeId is the node's InfiniBand GUID as described in the previous paragraph.

5.3.4 Cray XT3

Our second component to implement the Fountain server's `DataSource` policy is a module capable of monitoring the resources of a Cray XT3 parallel computer. The Cray XT3 is a massively parallel processing (MPP) system that was jointly developed by Cray Inc. and Sandia National Laboratories [33]. An individual Cray XT3 can scale from as few as 200 processors, to as many as 30,000. An XT3 system scales to larger configurations by using additional compute nodes to run user applications, and service nodes to provide management services. The systems software provided by Cray for the XT3 provides a rich set of features allowing a systems administrator to view and manage the XT3 as a single system image.

We have developed a module for the Fountain server that can obtain the total number of compute nodes and the number of available compute nodes and present this information to the cluster scheduler. This module was primarily developed as a means to show how Fountain can be extended to monitor specialized parallel resources. This module is similar to the InfiniBand network module described previously. For the discovery method, Fountain parses the output from a Cray XT3 administrative utility that list the number of installed compute nodes. The update method parses output from another administrative utility that list the number of available compute nodes.

This module is not feature complete, it exists merely as a demonstration of the capabilities Fountain has to monitor specialized parallel architectures. Future work exists to properly test and implement this module, as well as extend it to other architectures such as the IBM BlueGene.

5.4 Fountain client Utilities

To interact with the Fountain server, we developed several client utilities. The main desire to create these utilities was to test the node monitoring interface provided by the Fountain server. This allows development to take place on a system where the cluster scheduler cannot be adequately tested, such as a desktop or laptop. Also, since the Fountain server and node daemons typically run without user interaction, a separate client utility is required to provide an administrative interface to the system. The `fountainQuery` client utility provides basic mechanisms to query the Fountain server about node information, it essentially simulates the cluster scheduler. The `fountainAdmin` utility provides a basic administrative interface to the Fountain server. Finally, the `fountainTreeTest` client utility was developed to test the tree topology used by the Fountain node daemons.

5.4.1 Fountain Query

The `fountainQuery` client utility exists to perform queries against the Fountain server's node monitor database. It was developed as a method to simulate the Fountain server's interface to the cluster scheduler It supports all of the monitoring metrics explained in section 4.1.1 and features the following command line arguments:

```
--state [[op] [up | down]] : Display if the node is up or down.
--cprocs [[op] [value]] : Display the number of configured processors.
--aprocs [[op] [value]] : Display the number of available processors.
--uprocs [[op] [value]] : Display the number of utilized processors.
--cmem [units] [[op] [value]] : Display the amount of configured  memory.
--fmem [units] [[op] [value]] : Display the amount of free memory.
--cswap [units] [[op] [value]] : Display the amount of configured swap space.
--fswap [units] [[op] [value]] : Display the amount of free swap space.
--cpu [[op] [value]] : Display a node's CPU usage ranging between 0.0 and 1.0.
--node [[op] [value]] : Return results matching only the specified node ID(s).
--network : Display a node's network resources.
where [units] is one of [KB | MB | GB | TB | PB]
where [op] is one of [eq | ne | lt | gt | le | ge | like]
where [value] is the value to compare against
```

Multiple arguments can be combined to fine-tune the results returned. For memory quantities, an optional units attribute can be specified to compare and return results using those units.

5.4.2 Fountain Admin

The `fountainAdmin` utility acts as an administrative interface to the Fountain server and the Fountain node daemons. It supports four different *modes* of operation: **admin**, **trace**, **set**, and **bomb**. The **admin** mode is used to shutdown the Fountain server, change its logging level, erase the node monitor database, or force the master Fountain daemon to rebuild the tree topology. The **trace** mode is used to send a trace request message to every Fountain node daemon and receive their responses. It is primary used as a debugging tool for system administrators to ensure the tree topology has correctly recovered from a failure. The **set** mode is used to change a node's state to `Unavailable` in the event it needs to be taken offline and should not run user jobs. The **bomb** mode is used to send a bomb request message to every Fountain node daemon. Upon receiving such a request, a Fountain daemon cleanly exits without closing its connections to neighboring Fountain daemons. This results in a forced tree recovery or rebuild, depending on how many Fountain daemons were "bombed". The **bomb** mode supports an optional parameter used to match a node's unique ID so a specific node or a group of nodes can be "bombed" using a regular expression.

Since this utility can modify certain parameters that can be detrimental to Fountain's performance, it only accepts commands from a list of users in the Fountain configuration file. For a production system, this file should be readable and writable only by the root user.

5.4.3 Fountain Tree Test

The `fountainTreeTest` utility is only built when Fountain's debug mode is enabled through the configure script. It's a tool used to test the tree establishment, recovery, and rebuilding algorithms described in section 4.2. It does this by sending a bomb request message to specific nodes in the tree topology and checking the resulting topology after a recovery event. If the resulting topology differs from what is expected, this utility will log the failure so it can be investigated further.

CHAPTER 6. RESULTS AND ANALYSIS

In this chapter we present and discuss results from testing Fountain on 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. To achieve larger configurations than 34 Fountain daemons on 4pack or 65 Fountain daemons on Scink, multiple slave Fountain daemons were run on a each node in the cluster. The results in this chapter will discuss Fountain's performance querying various configurations of Fountain daemons, as well as the time elapsed during the tree recovery and rebuilding algorithms. Lastly, a Fountain daemon's node overhead will be quantified in terms of CPU usage and network bandwidth.

6.1 Query Performance

In section 1.2 we presented Fountain's the three design goals; to be fault tolerant, have a low per-node overhead, and be able to scale into the thousand node domain. Querying each node in the cluster is important to achieve the third goal of scalability since this time will only increase as cluster sizes become larger. To accomplish this goal, the Fountain daemons are arranged in a tree topology as described in Chapter 4. The results from performing node queries on a variety of Fountain configurations are shown in Tables 6.1 and 6.2. Figures 6.1 and 6.2 show a graph of these two tables. The time represented for each query is measured as the time it takes the Fountain server to send the master Fountain daemon 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 6.1 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 6.2 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 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. Fountain is able to query 265 daemons running on 4pack in around half of a second, while querying 1,025 daemons running on Scink requires just under one second. It is important to realize that Fountain is not intended exhibit raw speed in terms of query performance. It's design goals are primarily fault tolerance and scalability. If monitoring speed is desirable, Fountain can be extended to

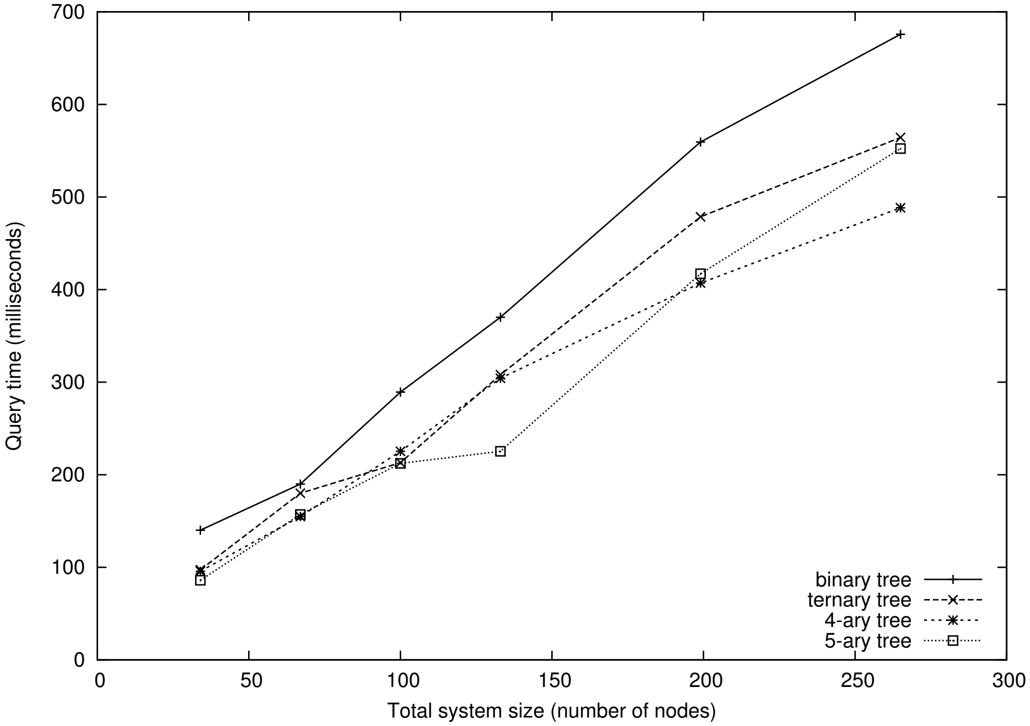


Figure 6.1 Elapsed time to perform a node query on 4pack

act as a wrapper around a more specialized monitoring component like Supermon [27], and present its data using the SSS interface.

The 4-ary and 5-ary tree topologies perform node queries faster than the binary and ternary trees because the depth of the topology is smaller, which is expected. In those configurations, each Fountain daemon is able to forward the query request to to more child daemons, which happens nearly in parallel.

6.2 Recovery Performance

The Fountain tree topology recovery algorithm attempts to recovery the tree topology from single node failures by replacing the failed node with another node. The time to perform this recovery is the time that elapses from when a neighboring node first reports the failure until

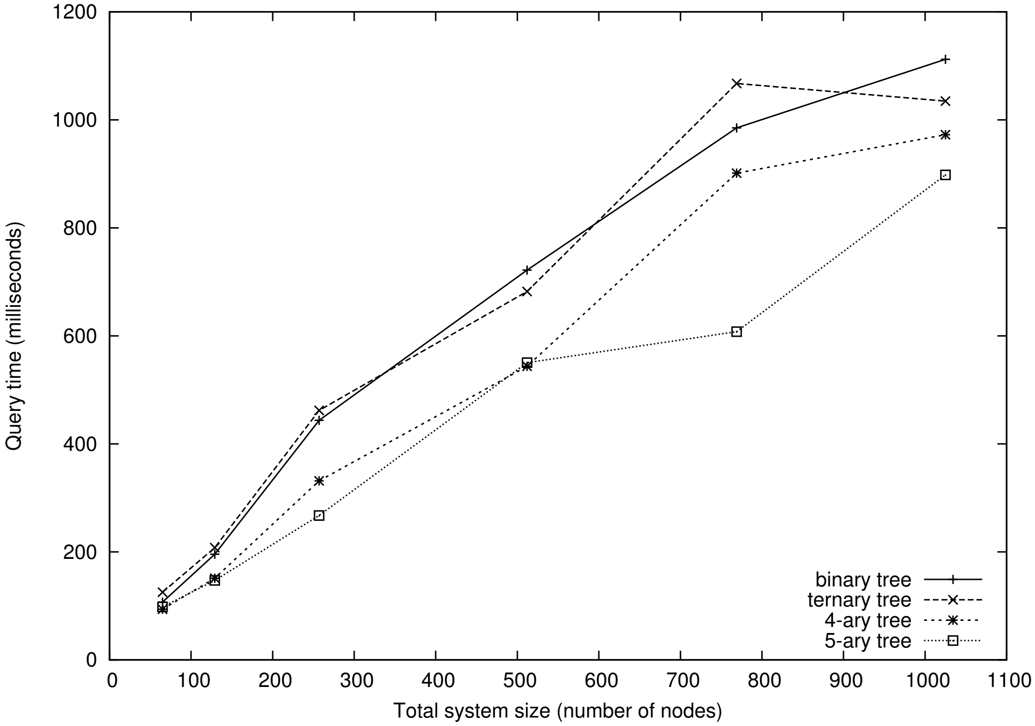


Figure 6.2 Elapsed time to perform a node query on Scink

a replacement node has been contacted and has successfully replaced the failed node. This includes the time it takes for all neighbors of the failed node to report the failure. Consequently, we expect tree topology configurations with a larger degree will exhibit longer recovery times. It's important for the recovery performance to exhibit good scalability since the failure rate of nodes tends to rise as cluster size increases. Therefore, we designed our tree recovery algorithm so it could recover from a single node failure by affecting the minimum number of nodes possible.

Table 6.3 and Figure 6.3 shows the average time to recover the tree topology from a single deterministic node failure. Table 6.4 and Figure 6.4 shows the average time to recover the tree topology from multiple concurrent node failures. 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 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 failed node that was reported first, will it respond to requests for another lost node and begin to recover the tree topology again.

Table 6.3 Elapsed tree topology recovery time from a single deterministic node failure (milliseconds)

System	Size	Binary	Ternary	4-ary	5-ary
4pack	34	95.23	167.66	195.74	230.76
Scink	65	127.47	173.66	220.11	275.69
Scink	129	145.74	179.68	235.40	282.26
Scink	257	136.15	180.71	227.91	278.72
Scink	513	139.33	183.68	236.37	284.21

Table 6.4 Elapsed tree topology recovery time from multiple node failures (seconds)

System	Number Failed	Size	Binary	Ternary	4-ary	5-ary
4pack	2	34	0.22	0.35	0.49	0.49
4pack	3	34	1.24	1.36	1.05	0.72
4pack	4	34	1.36	2.3	2.40	2.638
4pack	5	67	0.748	1.85	2.66	2.33
Scink	2	33	0.28	0.67	0.71	0.88
Scink	3	33	0.44	0.71	1.01	1.23
Scink	4	257	0.53	2.20	1.54	1.94
Scink	5	513	1.21	2.08	3.70	3.15

These recovery times show that Fountain is capable of efficiently reconstructing its tree

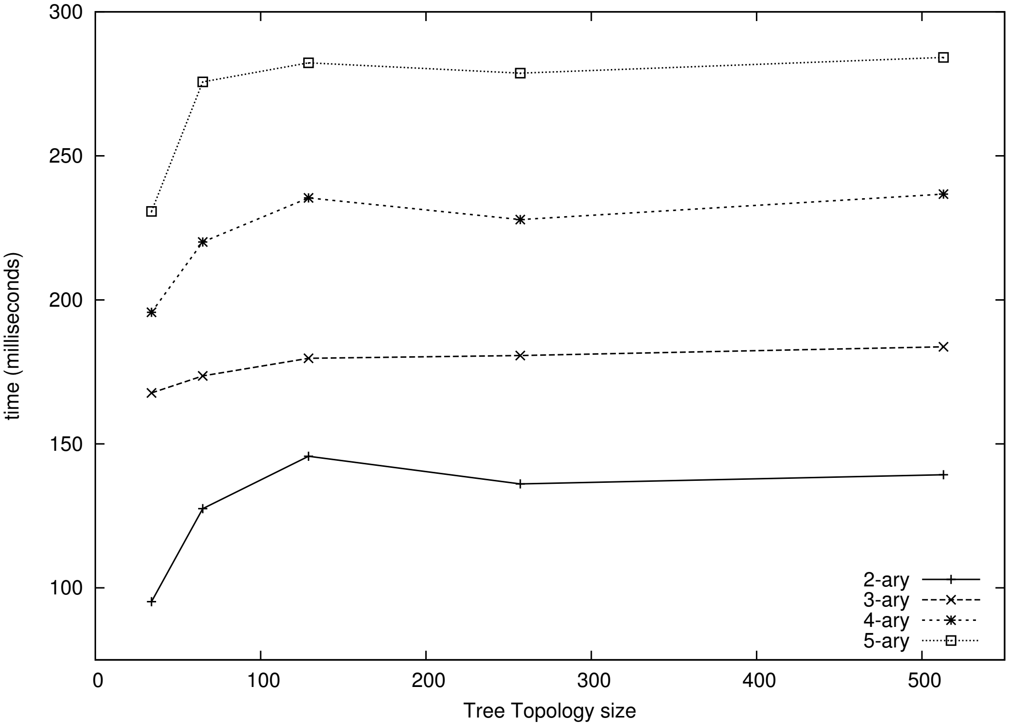


Figure 6.3 Elapsed tree topology recovery time from a single deterministic node failure (milliseconds)

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 nodes in the cluster. As an example, the recovery times for configurations of Fountain daemons ranging from 65 to 513 daemons on Scink only vary between 3 and 18 milliseconds when recovering from a single node failure.

6.3 Rebuild Performance

The tree rebuilding algorithm presented in section 4.2 is used as a last effort to reconstruct the tree topology if the master Fountain daemon determines it's not possible to recover the

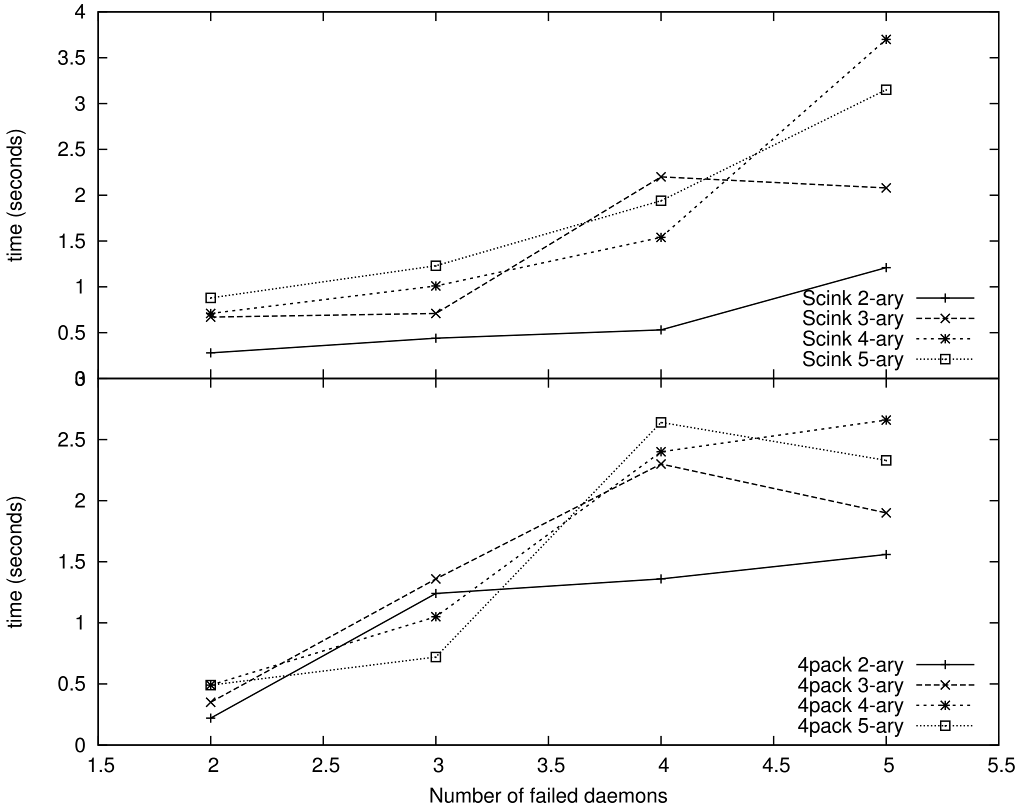


Figure 6.4 Elapsed tree topology recovery time from multiple node failures (seconds)

tree topology using the tree recovery algorithm. The numbers shown in Table 6.5 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 because each level in the tree will have progressively more nodes.

Figure **??** show the results from rebuilding 16 different tree topology configurations. The time required to rebuild the tree topology is significantly more than recovering from single or

Table 6.5 Elapsed tree topology rebuild time (seconds)

System	Size	Binary	Ternary	4-ary	5-ary
4pack	33	5.54	4.46	3.93	3.77
4pack	65	6.70	5.85	4.85	4.98
4pack	97	7.87	6.46	6.56	6.01
4pack	161	9.94	9.02	8.72	8.61
Scink	33	5.42	4.79	4.01	3.67
Scink	65	7.03	5.54	5.09	4.96
Scink	98	7.24	6.98	6.14	6.01
Scink	129	> 10	8.45	8.19	8.27

multiple failures. This is because each Fountain daemon has to incur the overhead of setting up a connection to the master daemon to report their lost parent node, then initiate the tree establishment algorithm. This results in a total of three connections for each Fountain daemon, and a total of nearly 200 connections when rebuilding a tree topology of 65 Fountain daemons on Scink. The recovery algorithm however, uses a maximum number of connections equal to one plus the degree of the tree topology. One each for the lost node's neighbors, one for the master daemon to connect to the replacement node, and one for the replacement daemon to connect to the lost node's parent.

In our current implementation, we use a fixed maximum time limit to transition from the tree topology rebuild state to the idle state (see Figure 4.3). This is because its unknown how many nodes in the cluster have failed when Fountain enters the rebuild state. If the number was known, the timer could be omitted and Fountain could transition the tree to the idle state only after all the non-failed nodes have reconnected. Another alternative to using a static timer value would be to create a formula based on the results shown in Table 6.5. The inputs to this formula would be the total number of Fountain daemons in the tree topology before entering the rebuild state, as well as the degree of the tree. The output of this formula would be the maximum amount of time to stay in the tree rebuild state, before transitioning back to

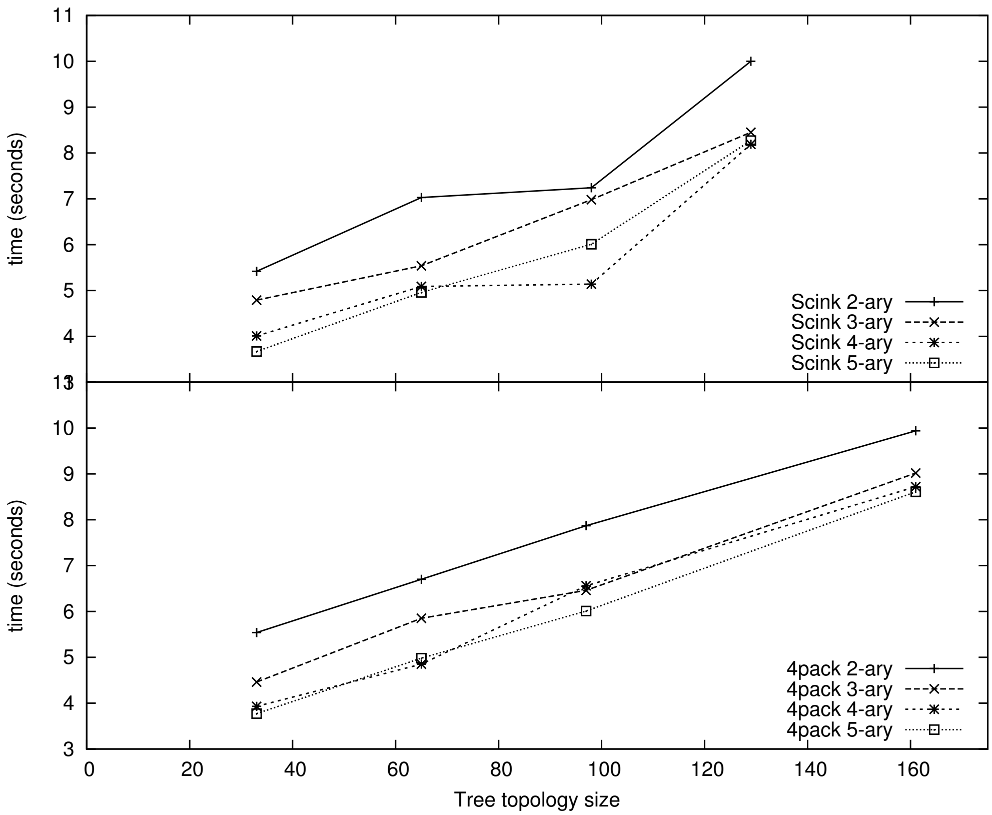


Figure 6.5 Elapsed tree topology rebuild time (seconds)

the idle state. However, the results from Table 6.5 are highly machine specific and may not be accurate for all Fountain installations.

6.4 Node Overhead

Our second design goal for Fountain was to have a low per-node overhead. This can be quantified in terms of CPU usage and network bandwidth required for a Fountain daemon. Ideally, Fountain would require zero quantities of both to perform its monitoring duties, which is clearly not possible. However, our design of Fountain strives to use a minimal amount of resources while accomplishing its goals as a node monitor. We are primarily concerned with node overhead on the cluster's compute nodes, since that is where user jobs are executed. Table 6.6 and Figure 6.5 summarize the node overhead for slave Fountain daemons running on

4pack. The monitoring bandwidth numbers were collected by running tcpstat [30] 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. A table and graph for node overhead statistics from Scink has been omitted since the numbers were essentially the same as shown is shown for 4pack.

Table 6.6 Node bandwidth usage on 4pack for a binary tree topology

Query Interval	Level in Tree	Total Nodes	Bandwidth (KB/sec)
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 bandwidth numbers shown in Table 6.6 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 has the most children and grandchildren. 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 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 at a regular interval 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 would cause 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.

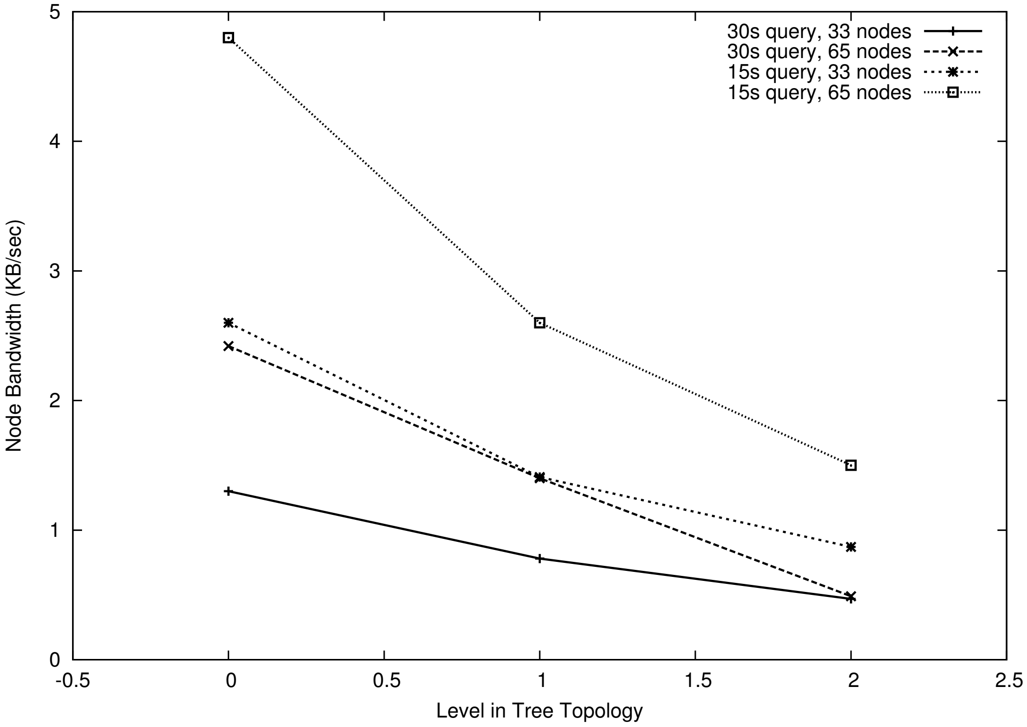


Figure 6.6 Node bandwidth usage on 4pack for a binary tree topology

6.5 InfiniBand Network Discovery and Polling

As described in section 5.3.3, the Fountain server is capable of discovering an InfiniBand network and periodically polling each discovered node for port counter information. A discovery event occurs once during the Fountain server's initialization process, and can be triggered when requested by the user. Polling each discovered InfiniBand node occurs at the same query interval rate used by the Fountain server to query the Fountain daemons. Table 6.7 shows the elapsed time to discover and monitor two InfiniBand networks.

The first network we tested is hal2004, a cluster of 64 dual 3.06 GHz Intel Xeon nodes connected to a 96 port Mellanox InfiniBand switch. Fountain reports a large number of switches when discovering this network because from a software perspective, the switch consists of many 8 port switch chips connected together to form a larger switch. Each individual switch chip has a global `system image GUID` which Fountain uses to present each switch chip as a child of one large switch. Appendix D shows an example `Node` object in a query response that contains an InfiniBand switch. We have left it up to the consumers of this data to interpret the individual switch chips, and their port counter information, as they please.

The second InfiniBand network we have tested Fountain with is a heterogeneous network in the Scalable Computing Laboratory (SCL). This network consists of several IBM Power5 eServers and dual AMD Opteron servers to test PVFS over InfiniBand [11]. While this network consists of less HCAs than hal2004, it uses slightly older switch hardware with older firmware and therefore the discovery time takes a bit longer.

Table 6.7 Elapsed time to discover and update an InfiniBand network (milliseconds)

System	HCAs	Switches	Discovery time	Update time
hal2004	64	54	781.8	35.5
SCL	12	7	4145.9	2.45

The results from discovering and polling these two InfiniBand networks shows the overhead is minimal. In a fairly heterogeneous network with older hardware such as the one used inside the SCL, the discovery time is slightly over 4 seconds while the time to poll each of the 19 nodes is a mere 2.5 milliseconds. The somewhat lengthy discovery time is acceptable since this process typically only happens once, while the polling process happens at a regular interval. On the more robust network used by the hal2004 cluster, the discovery process takes less than one second to discover each of the 64 nodes. In our current implementation the polling interval is fairly modest at around 30 seconds. Further work exists to increase this polling interval to provide more real time visualization opportunities. Care should be taken to avoid unnecessary perturbation of the InfiniBand network however.

6.6 Handling Client Queries

The end result of developing the Fountain node monitoring component is to provide its node monitoring data to external clients such as the cluster scheduler. If Fountain requires an extraordinary amount of time to respond to even simple client requests, it will not be a feasible component in the SSS environment. A variety of factors account for the amount of time Fountain spends processing a client query request. The number of entries in the node monitor database is a key factor because each entry has to be compared to see if it matches what information the client has requested. The content of the client request is also important because it will determine the size of the response message, which is often quite a bit larger than the actual client request message. If the request asks for the node state for a single node, the response will be much smaller than a request that asks for node state, configured memory, available memory, and any network information for all nodes.

The results from performing various client queries on node monitor database sizes ranging from 64 entries to 10,064 entries is shown in Table 6.8. Since the cluster Scink only has 64 compute nodes, we artificially increased the size of the node monitor database by writing up to 10,000 unique entries in the node list file read by the Fountain server during its initialization step. The two time metrics measured for handling client queries is the time taken to handle the query, and the time taken to send the response. The time taken to handle the query is

Table 6.8 Client Query Performance from Scink (milliseconds)

Size	Query content	Time to handle	Time to send response
64	node eq s11	0.56	0.58
64	state eq Up	5.04	6.44
64	all	24.66	23.21
64	node like host[0-4]	1.13	1.02
1,064	node eq s11	4.74	0.57
1,064	state eq Up	9.71	6.49
1,064	all	297.28	274.51
1,064	node like host[0-4]	13.87	7.60
5,064	node eq s11	21.45	0.57
5,064	state eq Up	28.13	6.44
5,064	all	1,382.35	1,269.76
5,064	node like host[0-4]	261.65	282.21
10,064	node eq s11	42.17	0.57
10,064	state eq Up	50.67	6.47
10,064	all	2,732.11	2,513.06
10,064	node like host[0-4]	118.81	56.61

somewhat difficult to quantify because it is composed of multiple facets, such as XML parsing, iterating through the node monitor database, and comparing various quantities. We expect the time taken to parse the query request will scale linearly with the node monitor database size, and the time taken to send the query response will scale linearly with the content of the client request. Four different queries were performed:

1. `node eq s11` - to match a single node
2. `state eq Up` - to match all nodes with a state of `Up`
3. `all` - to match all nodes and return all their statistics
4. `node like host[0-4]` - to match up to 10% of the node monitor database

For all cases, a set of default values is returned for each entry matched in the query. In addition to the default values, the second and third queries request other information that adds to the size of the response message, and consequently, the time required to transmit it.

As expected, the time to transmit a very large query response message containing 10,065 node elements and their statistics requires around 2.5 seconds, while handling this request requires slightly more than 2.7 seconds for a total of 5.2 seconds. We feel this number is acceptable since a node monitor database size of 10,064 entries will probably be quite rare for Fountain given that only a handful of installation sites on the top 500 supercomputing list have a commodity cluster with over 5,000 nodes [31]. For a more reasonable node monitor database size of 1,065 entries the time to handle and response to a query requesting all elements requires slightly over 0.5 seconds.

6.7 Improving Performance of the Node Monitor Database

As described in Chapter 5, the Fountain server uses a `map` container from the C++ Standard Template Library (STL) to maintain information about every node in a cluster. A `map` container by default sorts its elements according to a sorting criterion that is used for the actual key [14]. This container was chosen because it exhibits two qualities needed for the node monitor database:

- Each entry is a key/value pair, which can be user defined types
- Each entry in the `map` is required to be unique. We use a node's hostname as the key.

A typical usage pattern of our node monitor database involves inserting elements once, deleting them almost never, and accessing them frequently. According to Matt Austern, the standard associative containers may paint a slightly narrow view in terms of its applicability for our purposes [6]. In particular, a sorted `vector` container combined with one of the STL's built-in searching algorithms may perform better in terms of time and space than a `map` container [2]. This happens when the number of container accesses is larger than its number of insertions, which is the case with our node monitor database.

Our motivation to improve performance of accessing the node monitor database did not come from any profiling of the Fountain server; its performance is more than adequate based on our results in the previous section. We decided to explore this idea because the Loki library [2] provides a sorted `vector` container called an `AssocVector`, which is a drop in replacement for the STL's `map` container. Measuring its improvement, if any, is fairly trivial. Its disadvantages are linear time insertions and deletions, which isn't a concern for our node monitor database because they happen infrequently.

Table 6.9 shows the performance of the node monitor database in two cases. The first case is parsing the query response from the Fountain daemons. This requires the Fountain server to use the container's `find` method to locate each entry in the node monitor database that is present in the response message, then update its information. The second case is responding to a client query request where the Fountain server has to iterate over the entire node monitor database and compare the client's request with each entry. As described in the previous section, the size of the node monitor database was artificially increased by adding entries to the node list file. The times shown in Table 6.9 do not include the time it takes to receive and send the request and response messages along the socket.

Table 6.9 Node monitor database container comparison from Scink (milliseconds)

Container	Size	Node query time	Client query time
`map`	64	1.99	4.66
`AssocVector`	64	1.97	4.93
`map`	1,064	2.00	11.06
`AssocVector`	1,064	1.97	11.03
`map`	5,064	2.02	36.55
`AssocVector`	5,064	2.01	36.57
`map`	10,064	2.03	69.42
`AssocVector`	10,064	2.00	68.92

The results show the `AssocVector` container offers a very slight improvement over a `map` container when iterating over the entire node monitor database for both Fountain daemon queries, and client queries. The client query time is significantly longer than the time to process a node query because each entry in the database is compared with the client's request. The node query time stays nearly constant for all node monitor database sizes because the node query response from the Fountain daemons is a constant size of 64 entries when running on Scink. When parsing this response, the Fountain server uses the container's `find` method to update its information in the node monitor database, which is a fairly fast operation, requiring only 2 milliseconds on a container of over 10,000 entries.

In conclusion, the very small improvement in speed is probably not worth the risk of using the `AssocVector` container when the `map` container from the C++ STL has been widely used and tested.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

This thesis has described the design of Fountain, a node monitoring component implementing the Scalable Systems Software node monitor specification. It utilizes a rigid topology of daemons running on each compute node in the cluster to accomplish its monitoring goals while exhibiting favorable scalability as the number of nodes increases into the thousand node domain. Our major research contribution is the concept of maintaining a topology of persistent daemons, which is fairly novel in the area of node monitoring since most prior works have relied on multicast or transient connections and heartbeats to maintain node status.

As the cluster size increases, the node failure rate is expected to increase as well. Fountain adapts to node failures by recovering the tree topology and affecting the minimum number of nodes possible. The results from recovering from both individual and multiple node failures depend on the degree of the tree topology, and not the total number of nodes in the cluster. Lastly, we have demonstrated Fountain's extensibility as a cluster monitoring solution by developing a module to monitor the performance of an Infiniband network. Other solutions to push Fountain's monitoring capabilities into other domains exist in areas such as specialized parallel architectures like the Cray XT3 and IBM BlueGene.

7.2 Future Work

A few areas exist to improve upon Fountain's design. Most notable, the use of threads should be investigated for certain components where it may be advantageous such as the master Fountain daemon. It could use a thread for maintaining the tree topology, and a separate thread for handling query requests from the Fountain server. This may prove to be beneficial if the Fountain server's query interval is increased greatly. The Fountain server could also utilize either a thread pool or a thread per connection model for servicing client requests in larger cluster environments.

The use of non-blocking sockets should also be investigated at some point in the future. While our current implementation utilizing blocking system calls to read and write data on sockets is not problematic, it poses certain design restrictions such as the use of `pulse` messages between the Fountain daemons. This problem may be more effectively eliminated with the use of non-blocking sockets and timeout intervals.

7.2.1 Fountain Daemons

Our current implementation of Fountain relies on a single master daemon to coordinate the tree topology of slave Fountain daemons. This setup has worked well in our testing of configurations up to 1,025 daemons. However, the use of a single master daemon may prove to be troublesome as Fountain is deployed on larger clusters. It may prove useful to introduce the concept of sub-master Fountain daemons at certain levels in the tree topology. In such a system, each sub-master daemon would be responsible for maintaining a tree topology of a portion of the entire topology. The master daemon would be responsible for coordinating the sub-master daemons together. This concept is somewhat similar to how Ganglia achieves its federation by utilizing multiple `gmetad` processes in a cluster [17]. Using multiple sub-master daemons would lessen the load on a single master daemon in terms of recovering from node failures.

The current algorithm used to form the tree topology of Fountain daemons builds a n-ary breadth first tree where each level is completely full except the bottom level. In larger environments, the bandwidth usage of the daemons closer to the root of the topology may cause undesired node perturbation. Thus, It may become necessary to implement a more advanced algorithm for establishing the tree topology where the underlying network topology is considered. The tree establishment algorithm also uses a fixed tree topology degree. If the algorithm is enhanced to become aware of the physical network topology, it may be advantageous to vary the tree topology degree depending on the level of each Fountain daemon.

7.2.2 Fountain Server

The Infiniband network monitoring module explained in section 5.3.3 shows how Fountain is an extensible monitoring system capable of monitoring both node specific and server specific resources. Further work exists to expand this extensibility into domains such as parallel file systems, and additional network connectivity such as ethernet, and Myrinet. Fountain should also be improved to handle multiple networks for a single cluster rather than its current limit of a single network configuration.

The specialized monitoring tools mentioned in Chapter 2 may offer certain information that is desirable by other components in the SSS environment. Rather than implementing a Fountain module to duplicate this behavior, it maybe more feasible to create a simple wrapper around the component and use Fountain to present the information in the already familiar SSSRMAP specification for other SSS components. Using this method would require minimal work for Fountain and very little change to the SSS components requesting this information.

7.2.3 Graphical User Interface

A good monitoring tool should have an easy to use interface for both its users and administrators, which our current implementation is lacking. A prototype GUI application for an administrative front end to Fountain was shown in Figure 5.9. When completed, this project will allow a system administrator to view the status of multiple clusters quickly and easily. Historical system usage graphs, network visualization, and job status are features that will be integrated into this component.

APPENDIX A. Verification of Node Monitoring Algorithms using SPIN

Introduction

This appendix describes the process of using the model checker SPIN to verify the Fountain tree establishment algorithm presented in section 4.2. We will show results from expressing this algorithm using the high level Promela (PROcess MEta LAnguage) programming language and executing both simulation and verification runs. Verification of this algorithm is important since the other algorithms used by Fountain to maintain the tree topology depend on an accurate tree topology to function. In this sense, verification of the model means it is free of deadlocks, assertion violations, and race conditions.

The reason for using SPIN for this project is twofold. First, it is both freely available and an extremely powerful model checker. It has been used to verify the safety and security properties of numerous algorithms, including those used in operating systems, communication networks, and railway signaling protocols [24]. Secondly, SPIN was chosen for this project is because the Promela constructs provided by SPIN map easily to the facilities needed by a distributed system like the Fountain node monitor. Remote processes can be modeled by the proctype declaration, and sockets for wire protocol communication can be modeled by channels. Lastly, SPIN's simulation and verification modes provide two powerful methods for both designing and ensuring the correctness of a distributed algorithm. The simulation mode provides an easy method to debug and design a distributed algorithm like the Fountain tree establishment algorithm. Without SPIN's simulation mode, designing distributed algorithms is very difficult and often limited to using expensive debuggers or parsing through log files by hand. SPIN's verification mode provides an easy way to ensure the algorithms modeled are both deadlock free and correct.

Modeling Fountain using Promela

This project started with the mindset that it may not be possible to accurately model or verify the Fountain daemon monitoring component due to the state space explosion problem when modeling systems of distributed processes. However, previous experience with the current implementation of Fountain indicates that deadlocks or bugs in the tree establishment algorithm will manifest themselves even when simulating or verifying a small number of nodes. While initially performing simulation and verification runs on small system models, more advanced concepts of Promela and SPIN may allow verification of larger models as this project matures.

Similar to the authors of [18], each Fountain daemon can be considered to have three separate layers for modeling purposes. The lowest layer is the operating system layer representing operating system services. These services include all the system calls, including the socket system calls such as `bind`, `accept`, `read`, `write`, and `select`. The second layer is the message handling layer which executes commands received by a Fountain daemon. The third and topmost layer is the logic layer which is responsible for initialization of a Fountain daemon.

This three tiered modeling approach maps easily to the constructs provided by Promela. For the lowest layer, the UNIX socket Promela model from [18] was chosen since this allowed a preexisting approach to be used. Some small changes were made to the inline `read` function since the authors of the MPD version had the requirement of no blocking calls during the message handling code. The only blocking system call they used was a call to `select` in the main loop. The tree establishment algorithm used by Fountain requires a `read` blocking system call because the master daemon needs to receive a `join_ack` response from the slave daemon before it's added to the tree toplogy. Without the blocking `read` system call, this would require all socket communication to be proceeded by a system call to `select` first, which may be desirable for the authors of [18] to model their MPD system, but in our case with Fountain, it is not desirable.

The second layer for handling messages consists of two inline Promela functions. One to handle connection requests, and one to handle messages on existing connections. The third layer for initialization logic is handled by Promela's built in constructs for creating processes.

Table A.1 lists all the Promela files used to model the various components for Fountain. The bulk of the tree establishment algorithm is contained in the main.pr and handlers.pr files. The global data structures and definitions in the file fountain_header.pr are an important concept for modeling Fountain. The `conn_info` variable is an array of `conn_info_type` structures. Each `conn_info_type` structure holds a variable to represent the file descriptor of the remote socket, the process ID of the process that owns this `conn_info_type` structure, and the use flag. The use flag has 7 possible values: `FREE`, `NEW`, `PARENT`, `LCHILD`, `RCHILD`, `AWAIT_ACCEPT`, and `NONE`. The use flag tells the owning process how to handle an incoming message on that particular socket. Initially, all sockets in the `conn_info` array begin with a use flag of `FREE` that indicates that particular socket has not been allocated. After a process calls `connect` two sockets are allocated, a local one and a remote one. The local socket is owned by the process ID that made the call to `connect` and the remote socket is owned by the process ID who the local process is trying to connect to. The local socket's use flag is initially set to `NONE` and the remote socket's use flag is initially set to `AWAIT_ACCEPT`. After the remote process realizes a connection attempt has been made, they will accept the connection request and change the use flag to either `LCHILD` or `RCHILD`.

Tree Establishment Algorithm in Promela

The Promela code for the master Fountain daemon is shown in Figure A.1. The master Fountain daemon first adds itself to the global `tree_topology` array then launches a predetermined number of slave Fountain daemon processes. Then it launches the process to check the resulting tree topology and lastly enters its main loop where it services incoming connection requests and existing connections. The bulk of the tree establishment algorithm is modeled using the message handling code where the master Fountain daemon responds to join messages with a join accept message indicating which parent Fountain daemon the slave Fountain daemon should join.

When the master Fountain daemon receives a connection request, it calls the `handle_listener`

Table A.1 Listing of Promela code used in this project

Filename	**Lines**	**Purpose**
all.pr	1	Number of Fountain daemons to simulate or verify
checker.pr	72	tree_check Proctype to verify the tree topology is correct
fountain_header.pr	159	Global variable declarations and #define statements
handlers.pr	155	Message handling
main.pr	196	Main code for launching Fountain Promela processes
sockets.pr	138	Promela socket implementation from [18]

inline function. Inside this function, it iterates over the `tree_topology` array to find the first entry in the tree topology that has less than 2 children. The number of children per node in the tree topology is represented by a `num_children` variable. After selecting an available parent daemon, the master daemon responds to the slave daemon with a join accept message and indicates which parent daemon to join. The master daemon then executes a blocking read call and waits for a join acknowledgment message from the slave daemon. After receiving the join acknowledgment, the tree topology array is updated by appending the newly joined node to the end of the array and updating the `num_children` variable for its parent. Figure A.2 shows how the `tree_topology` array represents the tree topology.

The slave Fountain daemon processes are very similar to the master Fountain daemon. Their main difference happens before they enter the main loop. Before entering the main loop, the slave daemons first send a join message to the master daemons, which responds with the appropriate parent daemon to join. The slave daemons then joins this parent daemon and responds to both with an acknowledgment message. The main loop of a slave daemon modeled in Promela is exactly the same as the master daemon. If it detects an incoming connection on

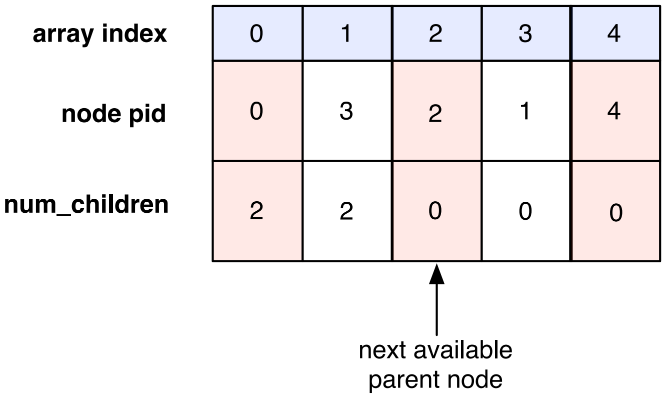
```
proctype master_fountain_node () {
        tree_topology[0].fd = _pid;
        availableTreePosition = 1;

        launch_slave_nodes();
        run tree_check();

        /* main loop */
        do
        :: prepare_for_select();
                end_select:
                select();
                do
                :: (n < CONN_MAX) ->
                        if
                        :: (readable_fd(n, _pid)) ->
                                handle_input(n)
                        :: (readable_lp(n, _pid)) ->
                                handle_listener(n)
                        :: else fi;
                        n = n + 1
                :: else -> break
                od;
        od
}
```

Figure A.1 Promela code for master Fountain proctype.

its listen port, it calls the `handle_listener` inline function just as the master daemon does. Inside that function there's an if statement that determines if the caller is the master daemon or a slave daemon. Slave daemons will respond to the join request with a join accept message if they have an available position for a child daemon. This is indicated by two per-process bit variables `haveLeftChild` and `haveRightChild`. Both bit variables are initially set to 0, and will be set to 1 when the node has that child. It is an assertion violation if a node attempts to join a parent daemon who has no available spots for a child node. Lastly, the slave daemons do not modify the global `tree_topology` array, they are added by the master daemon after successfully joining a parent daemon and sending a join ack message.



array index	0	1	2	3	4
node pid	0	3	2	1	4
num_children	2	2	0	0	0

Figure A.2 Fountain tree topology represented as an array.

Verification of a Correct Tree Topology

At first it may seem odd that the process to start checking the tree topology is started immediately after starting the slave Fountain processes. However, Promela provides a `timeout` statement which guards further execution until all other processes have no statements that can be executed. Essentially what this means is either all the other processes have reached a valid end state, or there is a deadlock. If a deadlock occurred, the process to check the tree topology will discover the tree is not valid. If all the other processes reached a valid end state, the tree checker process will correctly verify the tree topology. Essentially what the `tree_checker` process does is ensure that each child node has a parent, and each parent node has the correct number of children. If anything is found to be incorrect, an assertion will be violated and the SPIN verification will fail.

The tree checker process itself is quite simple. It iterates over the global `conn_info` array and checks each entry for correctness. Since each `conn_info` entry has a use flag and a field identifying the remote port its connected to, a correct entry in the array will satisfy the following properties:

```
(use_flag == PARENT) && (remote use_flag == LCHILD)

or

(use_flag == PARENT) && (remote use_flag == RCHILD)

or
```

```
(use_flag == RCHILD) && (remote use_flag == PARENT)
or
(use_flag == LCHILD) && (remote use_flag == PARENT)
```

If any entry in the array does not meet one of these properties, it is not properly connected in the tree topology and an assertion will be raised. This is important since slave Fountain daemons are instructed to join a parent Fountain daemon after connecting to the master daemon. If this parent daemon is not the master daemon, the slave daemon has to initiate another connection, thereby allocated two additional socket resources. After successfully connecting to its parent daemon, the slave daemon has no need for the socket resources previously allocated for its connection to the master daemon. Releasing these resources is important since it could otherwise cause the system to run out of memory. If the slave daemon doesn't release their unused sockets, the tree checker process will discover this and raise an assertion.

When running the `pan` verifier generated by SPIN the `-n` and `-q` runtime arguments were used. The `-n` argument omits a printout of states that are not reached at the conclusion of the exhaustive verification. There are certain states which will never be reached since they contain assertions indicating an incorrect or deadlocked tree establishment algorithm. The `-q` argument indicates that all message channels have to be empty for an end state to be valid. A process end state during verification occurs when a process has no other executable statements. Normally, an end state occurs when a process reaches the closing curly bracket of its defining body. However, in some protocols there's an infinite loop which will never allow the process to reach this closing curly bracket. To indicate a valid end state for the Fountain processes, a Promela end state label was added in the call to `select`.

Results and Future Work

Successful simulation of up to 5 Fountain daemon successfully joining a tree topology was achieved using the Promela models described thus far. Each simulation run was produced by giving SPIN the `-a` runtime argument. Each model simulated successfully without assertion or end state errors. Verification on the other hand was much more difficult. Exhaustive verification was only possible on only a system of 3 Fountain daemon before the state vector and the memory requirements became too large. The `-DSAFETY` argument when compiling the `pan.c` file created when running SPIN in verification mode. This argument disables support for finding non-progress cycles during the verification, which is fine since non-progress cycles are not a concern in this project. The `-DCOLLAPSE` argument was also used to compress the state vector somewhat. When performing exhaustive verification on a model of 4 Fountain daemon, the `pan` process ran for about 90 minutes and used nearly 4 gigabytes of memory until it was killed by the kernel. Thus, for systems of 4 and 5 Fountain daemons the `-DBITSTATE` compile time argument was used instead of `-DCOLLAPSE`. This option uses supertrace exploration instead of exhaustive exploration, so while it is more efficient in terms of execution time and memory usage it does not exhaustively verify the model.

The verification results from systems of 1, 2, 3, 4, and 5 Fountain daemons is shown in Table A.2. The execution time to exhaustively verify each model, as well as the memory requirements and state vector size is shown. The memory requirements are what is reported by the `pan` verification executable produced by SPIN when the verification completes. The states stored column indicates the size of the state vector for that model, where each state in the vector occupies the number of bytes in the vector size column. Clearly, as the model size increases, both the state vector and state size increase the memory requirements for verification dramatically.

Table A.2 Verification statistics for the tree establishment algorithm

Model Size	**Time (s)**	**Memory (MB)**	**Vector Size (bytes)**	**States Stored**	**States Matched**	**Search Depth**
1	0.00	4.982	52	21	0	20
2	0.01	4.982	116	1753	569	188
3	9.54	44.406	172	111,450	476,996	466
4[1]	14.06	1114.82	316	339,886	199,268	947
5[1]	20.35	1602.76	404	385,279	281,538	1747

The verification runs were performed on a dual AMD Opteron system with each processor

[1] -DBITSTATE compile time argument used

running at a 1.5 GHz clock speed. The system has a total of 7.5 gigabytes of physical memory, and runs Debian Linux. The timings reported in the second column of A.2 were obtained using the build in shell command `time`. CPU usage was near idle at the time the verification runs were started.

When exhaustively verifying the tree establishment algorithm, no deadlocks, race conditions, or assertion violations were discovered. This is not unexpected since the current implementation of Fountain has been used successfully for almost a year on production clusters in the Scalable Computing Laboratory. As shown in Table A.2 the memory requirements and execution time to exhaustively verify systems of 1, 2, and 3 Fountain daemons were very modest. The memory requirements for exhaustively verifying systems of 4 or 5 Fountain daemons were too large to finish. Clearly work remains in this area to reduce the state vector size and improve the verification time for systems larger than 3 Fountain daemons.

Verification of the tree establishment algorithm has provided us with the experience and insight needed to continue work in this area and verify other parallel algorithms used by Fountain. The tree recovery and rebuilding algorithms are two algorithms that would benefit from verification in the future. Similar to what the authors of [18] did to verify their ring reconstruction and barrier algorithms, other algorithms used by Fountain would most likely be easier to exhaustively verify by using a pre-constructed tree of Fountain daemons. This would eliminate much of the state vector space that is otherwise occupied with the messages passed between Fountain daemons for the tree establishment.

Conclusion

This appendix has described how to use the model checker SPIN to verify the binary tree establishment algorithm used by Fountain. SPIN's simulation mode allows the rapid prototyping of new distributed algorithms since most errors can be found even when simulating a handful of remote processes. The exhaustive verification mode provided by SPIN allows an easy method of checking the security and correctness properties of a distributed algorithm like the tree establishment algorithm used by Fountain. While only exhaustive verification was successfully performed on models of 3 Fountain daemons, using state vector compression techniques allowed the verification of larger system models. Work remains to reduce the state vector size as well as to verify other distributed algorithms used by Fountain.

The Promela source code used to verify the node joining algorithm described in this paper is available at `http://www.scl.ameslab.gov/~samm/fountain/`.

```
proctype slave_fountain_node () {
        connect(fd, 0);
        masterFountainNodefd = fd;
        make_join_msg(_pid);
        write(fd, msg);

        if
        :: read(masterFountainNodefd, msg) ->
                if
                :: (msg.cmd != join_accept) ->
                        assert(0)
                :: else ->
                        unsigned parentNode : FD_BITS;
                        parse_join_accept_msg(parentNode);
                        if
                        :: (parentNode == master_fd) ->
                                /* already connected */
                        :: else ->
                                /* connect to parent node */
                        fi;

                        /* set our parent connection */
                        set_handler(fd, PARENT)

                        /* respond to with join_ack
                        make_join_ack_msg(_pid);
                        write(masterFountainNodefd, msg);
                        if
                        :: (fd == masterFountainNodefd) ->
                                skip
                        :: else ->
                                close(masterFountainNodefd)
                        fi
                fi
        :: else fi;
        /* begin main loop */
}
```

Figure A.3 Promela code for slave Fountain proctype.

APPENDIX B. Moso Process Manager

Moso is a prototype process manager for the SSS environment. It was created to solve the disjoint relationship between the Fountain node monitor and MPD process manager, which is explained in detail in section 4.1. Its role as a process manager is critical since it provides the services to start, stop, and signal user processes as requested by other components such as the scheduler or queue manager. To accomplish this, Moso uses the concept of a process group, where a single process group consists of `n` processes running on `n` hosts in the cluster. This concept is somewhat similar to the POSIX definition of a process group, with the exception that a Moso process group is split across nodes in a cluster.

To manage user processes, Moso uses one Moso daemon process per node in the process group. The role of a Moso daemon is similar to that of the managers in the MPD process manager [7]. It collects standard output and error from the user process, forwards signals, and monitors the exit status. To collect standard output and error in a scalable fashion, the Moso daemons are arranged in a binary tree topology, shown in Figure B.1. The large black circles represent Fountain daemons, configured in a binary tree topology represented by the larger solid black lines. They can fork and manage multiple Moso daemons concurrently. The dashed vertical lines represent pipes used for communicate between a Fountain daemon and its forked Moso daemon. The smaller grey circles with numbers representing their process group are the Moso daemons. They each have a single pipe to their user process, represented by another vertical dashed line. The smaller solid grey lines between the Moso daemons represent their binary tree topology within their process group. In Figure B.1 there is a total of 7 Fountain daemons, 12 Moso daemons, and 3 process groups. Note that the master Fountain daemon is not capable of forking a Moso daemon in our design.

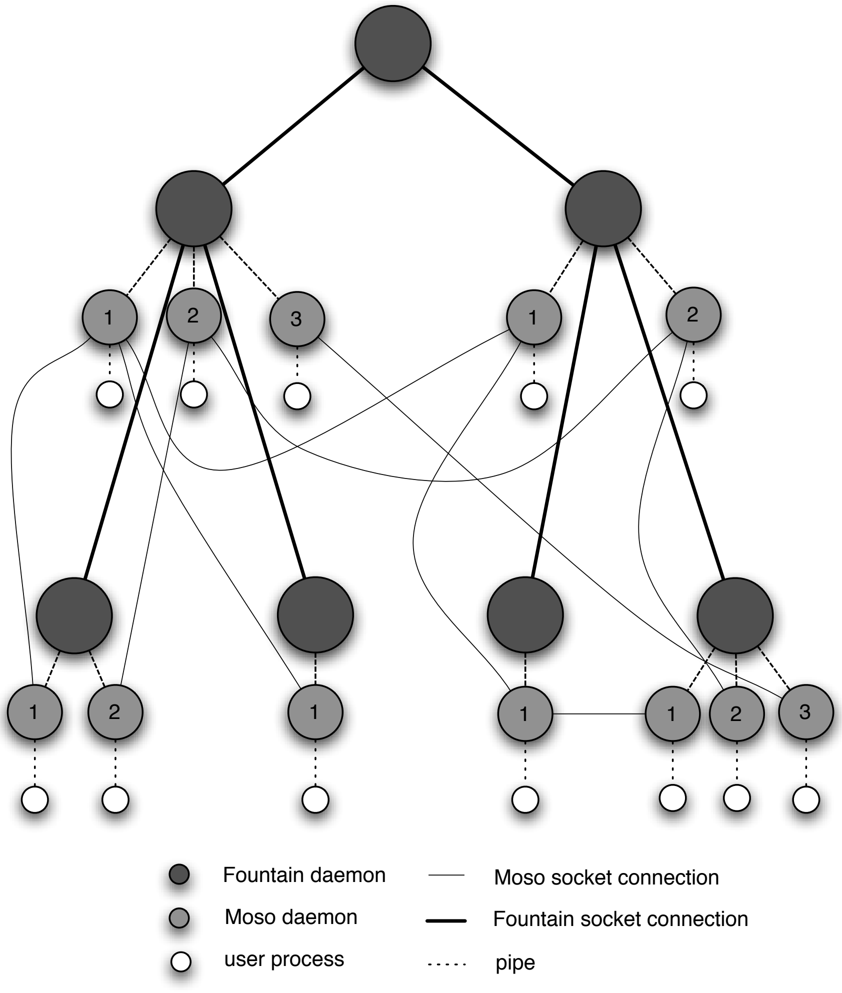


Figure B.1 Fountain daemons, Moso daemons, and user processes

A Moso daemon is created when a process group is created, exists for the lifetime of the process group, and terminates when the process group finishes. Since they only exist for the lifetime of their process group, Moso requires the help of Fountain to create and destroy Moso daemons in order to achieve decent scalability. Moso itself maintains an internal container of process group identifiers and their associated data, such as state, list of hosts, and any other pertinent information. The following sections will explain how Fountain handles requests from Moso to create, signal, and kill process groups.

Creating a Process Group

When the cluster scheduler decides its time to run a user's job, it will inform the queue manager, which will request a process group be created for that job. To maintain compatibility with other components in the SSS environment, Moso uses the Less Restrictive Syntax (LRS) for XML messages. The following is an example request to create a process group on 5 hosts:

```
<create-proces-group>
  <submitter>samm</submitter>
  <process-spec>
    <env name="PWD">/Users/samm/moso</env>
    <exec>hostname</exec>
  </process-spec>
  <host-spec>m0
  m1
  m2
  m4
  m5
  </host-spec>
</create-process-group>
```

When Moso receives such a request, it creates a process group object for internal bookkeeping purposes and forwards the request to the master Fountain daemon. Upon receiving the request, the master Fountain daemon performs the following tasks:

1. Check the contents of the `host-spec` element to ensure each hostname belongs to a valid host in the Fountain tree topology.

2. Translate the request into an SSSRMAP formatted message and send it to its child Fountain daemons.

3. Wait for responses from its child Fountain daemons indicating if they successfully forked a Moso daemon.

4. Collect Moso daemon listen ports from the response messages, assemble them into a list.

5. Send a message containing the list of Moso daemon listen ports to its child Fountain daemons.

6. Wait for responses from its child Fountain daemons. Inform Moso if the process group was created or not.

The first task in the above list already solves the disjointness problem we experienced with Fountain and the MPD process manager. If Moso attempts to create a process group with a host that does not exist in the Fountain tree topology, the master Fountain daemon will respond with an error. If all the hosts in the requested process group are found in the topology, we can assume each node is ready to run a user job. Additional work remains to coordinate the list of hosts in the tree topology with each node's state that is maintained by the Fountain server. A system administrator can set a node's state to `Unavailable` through the Fountain administrative interface. While it is possible a node with a state of `Unavailable` can have a Fountain daemon running on it, any requests to create a process group containing any host with a state of `Unavailable` should be rejected by the master Fountain daemon. In practice, this event should rarely happen since the queue manager will only request a process group be created after it has been told to do so by the cluster scheduler. The cluster scheduler will only schedule the user's job when all the requested hosts have an acceptable state.

When the slave Fountain daemons receive a create process group request, they first forward the request to their child Fountain daemons. Next, they match their hostname against the `host-spec` element. If a match is found, they fork a Moso daemon. If an error occurred during the fork system call (ex: process table full) an error response is sent to the Fountain daemon's parent. Before forking a Moso daemon, a Fountain daemon opens a listening port for it to inherit. The `fork` system call provides the semantics for child processes to inherit all open file descriptors of their parent. We chose this design because during testing it proved to be too time consuming to allow a Moso daemon to open its own listening port, then send this port number to its Fountain daemon. On several compute nodes under high load, we measured times in excess of 10 seconds to just fork a Moso daemon. This amount of time is insufficient for the master Fountain daemon's purposes, which uses timeout values while waiting for response messages from its children after sending them a create process group request.

After forking a Moso daemon, the Fountain daemon collects a listening port message from each of its children and appends their information into a message containing its Moso daemon's listening port. Eventually, the master Fountain daemon will receive all of these messages and collect them into a single message, then send it down the tree topology. This process is required because the forked Moso daemons need to know which parent daemon to connect to, and it is not always the same parent as its Fountain daemon (see Figure B.1). When the slave Fountain daemons receive the message containing all of the Moso daemon listening ports, they find their Moso daemon in the list and pick its parent node for the binary tree topology using equation B.1. They then send a success or failure response up the Fountain tree topology where the master daemon collects them.

$$parent = \lfloor (pos - 1)/2 \rfloor \tag{B.1}$$

After the master Fountain daemon receives all the responses from the slave daemons, it checks if the requested hosts in the `host-spec` element forked Moso daemons. If every host requested actually forked a Moso daemon, a success response is sent to Moso containing the list of hosts in the process group. Moso then adds this information into a process group container.

The act of starting the user job is controlled by the Moso daemons themselves. The root Moso daemon is aware of how many daemons will be present in its binary tree topology when it's fully formed, so it will periodically send a `trace` request message to its children. Its children will forward that request to their children, then collect their responses and add their own hostname before sending a trace response to the root Moso daemon. Only after the correct number of Moso daemons are found in the topology will the root Moso daemon send a `start` request message to its children. This procedure is shown in Figure B.2. It is somewhat similar to a `MPI_Barrier` collective used by most Message Passing Interface implementations.

The reason we chose to push the role of starting the user job into the domain of the Moso daemons is similar to our decision to have a Fountain daemon open its Moso daemon's listening port before forking it. It proved to be too time consuming during testing to wait until each forked Moso daemon had connected to its selected parent daemon. The time required to handle

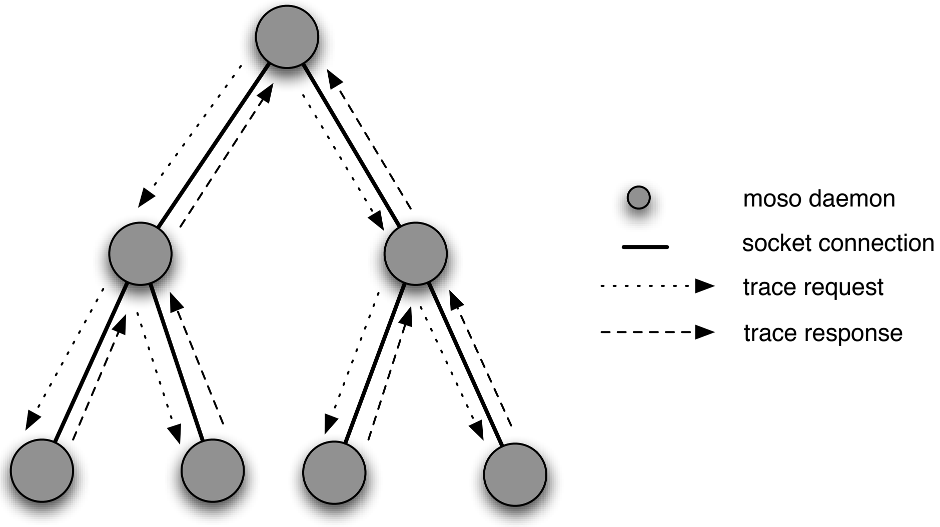


Figure B.2 Trace request and response messages sent by Moso daemons

a create process group request is much faster using these two methods.

Signaling a Process Group

When the queue manager has determined a user job has run over its allocated time limit, it will ask the process manager to deliver various signals to the process group. Some process manager implementations only deliver signals to the first process in the process group. They rely on the user job being responsible for propagating the signal to the other user processes. Moso's implementation sends the signal to each user process in the process group. The request received by Moso looks similar to this:

```
<signal-proces-group>
  <signal>SIGTERM</signal>
  <process-group>
    <pgid>2</pgid>
  </process-group>
</signal-process-group>
```

Moso performs some semantics checking on the request, such as enforcing a valid process group id, and a valid signal type, then sends the request to the master Fountain daemon. It does not expect a response, similar to how the UNIX `signal` command does not return a value, unless the process does not exist in which case it returns -1.

When the master Fountain daemon receives the signal process group request it translates the request into an SSSRMAP message and forwards it to its child daemons. The child daemons first forward the request to their children, then match the process group id to any Moso daemons they have forked. If a match is found, the signal is delivered. If no match is found, the request is ignored.

Killing a Process Group

A process group is killed after all of its client processes have exited. At that point, each Moso daemon has served its purpose and can be safely killed without affecting any other process. The problem here is detecting when all the user processes have exited. To accomplish this, the Moso server will periodically send a request message to query each Fountain daemon about its Moso daemons exit status. Only after all of the user processes in the process group have returned an exit status, will the Moso server send a kill process group request formatted like so:

```
<kill-proces-group>
  <process-group>
    <pgid>2</pgid>
  </process-group>
</signal-process-group>
```

Upon receiving this request, the master Fountain daemon translates it into an SSSRMAP message and forwards it to its children. The slave Fountain daemons first forward the request to their children, then match the process group id element to their list of Moso daemons. If a match is found, the Moso daemon is killed. Alternatively, Moso can kill a process group after being requested to do so by a user. The syntax for this request is similar to what is shown above.

Results and Discussion

We tested the Moso process manager on Scink, a cluster which is described in detail in Chapter 6. The Fountain daemons were arranged in a binary tree topology, with one, two, or four daemons running on each of the 64 compute nodes in the cluster. Measuring parallel job performance is a difficult task since it can depend on a great deal of parameters. For our purposes of assessing performance of a process manager, we desired to know the time it takes to create the process group, as well as the time it takes the Moso daemons to connect themselves into a binary tree topology.

The time taken to create a process group begins when the master Fountain daemon receives the request from Moso, processes it, and ends when it sends a success or failure response back to Moso. We expect this time to be somewhat similar to the time required for the Fountain server to query a similarly configured topology, except in this case we are forking another process (a Moso daemon) instead of collecting node monitoring information. Based on our results in Chapter 6, we expect this number to stay somewhat constant regardless of the number of hosts in the process group. This is because the create process group message has to be propagated to each Fountain daemon in the tree topology, even if the request only asks to create a process group consisting of a single host.

The time taken for the Moso daemons to bootstrap themselves into a binary tree topology begins when the root Moso daemon enters its main loop, and ends after it has determined the correct number of Moso daemons have joined. This time will probably vary wildly depending on the status of the hosts in the process group. If certain nodes in the cluster are running at near peak utilization, it may take their Moso daemon significantly longer to join the tree topology than nodes that are idle. Table B.1 and Figure B.3 show the results from testing Moso when creating four differently sized process groups on Scink. The numbers shown represent an average of three separate test runs.

The results are similar to what we expected. The time required to create a process group scales linearly with the number of Fountain daemons, requiring under 0.5 seconds to setup a process group of 64 hosts on all three configurations of Fountain daemons. The time to setup

Table B.1 Elapsed process group creation time on Scink

Fountain daemons	Hosts	Create Process Group (ms)	Bootstrap Binary Tree (sec)
65	3	280.71	6.26
65	9	285.84	7.46
65	27	301.03	13.46
65	64	315.93	23.80
129	3	354.54	5.64
129	9	353.75	7.40
129	27	360.71	19.23
129	63	426.71	23.49
257	3	403.52	5.79
257	9	437.21	30.67
257	27	404.12	30.59
257	63	458.27	22.60

a binary tree for the Moso daemons to communicate however, requires more time as the size of the process group is increased. This time seems to be somewhat random, and is probably related to the overhead of the Scink nodes in the process group. For example, when testing a 64 host process group created with 257 Fountain daemons, it required slightly over 20 seconds for the Moso daemons to create their tree topology. However, with the same configuration of Fountain daemons, it required over 30 seconds to create a process group of both 9 and 27 hosts. We expect this time to decrease when the compute nodes in the process group have utilization values closer to 0.

Summary

This appendix has shown how Fountain can provide the required process management services to a prototype SSS process manager called Moso. It accomplishes these tasks by forking Moso daemons to manage user processes. The Moso daemons in turn, use information from their parent Fountain daemon to bootstrap themselves into a binary tree topology, which

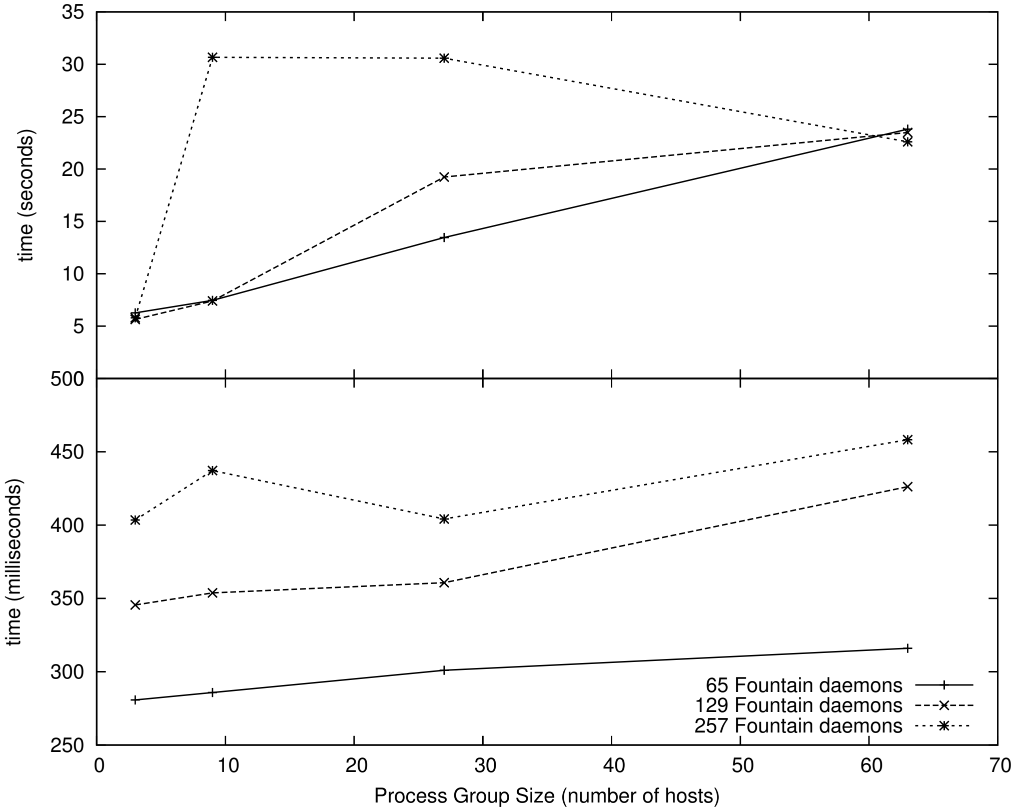


Figure B.3 Process group creation (lower) and binary tree (upper) on Scink

is used to forward standard output, error, and signals to and from the user processes in a scalable fashion. Once all the user processes in the process group have finished, the Moso daemons are killed to prevent unnecessary resource leaks. By combining Fountain and Moso together we have solved the disjoint node monitor and process manager problem experienced with the MPD process manager.

This project is by no means complete. Future work exists to properly incorporate advanced process management features, such as interactive jobs, attached debuggers, different executables for certain ranks, and incorporating parallel library functionality such as MPI.

APPENDIX C. XML Schemas

This appendix contains two XML schemas that specify the format of messages used by Fountain to interact with other components in the SSS environment. The actual process of checking if incoming and outgoing messages meet the requirements imposed by these schemas is called *validation*, which can be enabled or disabled with our chosen XML processing library [19]. An invalid XML message is one that does not meet the requirements imposed by its schema, such as:

- Valid element names and attributes, and their structure (ex: child elements)
- Valid element values (ex: string, positive integer, boolean, enumeration, etc.)

The SSSRMAP schema defines the Scalable Systems Software Resource Management and Accounting Protocol message format, while the Node Object schema defines the requirements for a Node element used to represent information about a node in a cluster. Fountain uses these two schema definitions to process and respond to node monitor requests from other components.

SSSRMAP XML Schema

```
<?xml version="1.0" encoding="utf-8"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:sssrmap="http://www.scidac.org/ScalableSystems/SSSRMAP"
  targetNamespace="http://www.scidac.org/ScalableSystems/SSSRMAP"
  elementFormDefault="qualified">
  <element name="Envelope" type="sssrmap:EnvelopeType" />
  <element name="Body" type="sssrmap:BodyType" />
  <element name="Request" type="sssrmap:RequestType" />
  <element name="Response" type="sssrmap:ResponseType" />
```

```
<element name="Object" type="sssrmap:ObjectType" />
<element name="Get" type="sssrmap:GetType" />
<element name="Set" type="sssrmap:SetType" />
<element name="Where" type="sssrmap:WhereType" />
<element name="Option" type="sssrmap:OptionType" />
<element name="Data" type="sssrmap:DataType" />
<element name="File" type="sssrmap:FileType" />
<element name="Count" type="positiveInteger" />
<element name="Status" type="sssrmap:StatusType" />
<element name="Value" type="sssrmap:StatusValueType" />
<element name="Code" type="sssrmap:CodeType" />
<element name="Message" type="string" />
<element name="SecurityToken" type="sssrmap:SecurityTokenType" />
<element name="Signature" type="sssrmap:SignatureType" />
<element name="DigestValue" type="sssrmap:DigestValueType" />
<element name="SignatureValue" type="sssrmap:SignatureValueType" />
<element name="EncryptedData" type="sssrmap:EncryptedDataType" />
<element name="EncryptedKey" type="sssrmap:EncryptedKeyType" />
<element name="CipherValue" type="sssrmap:CipherValueType" />
<complexType name="EnvelopeType">
  <choice minOccurs="1" maxOccurs="1">
    <choice minOccurs="1" maxOccurs="2">
      <element ref="sssrmap:Signature" minOccurs="0" maxOccurs="1" />
      <element ref="sssrmap:Body" minOccurs="1" maxOccurs="1" />
    </choice>
    <element ref="sssrmap:EncryptedData" minOccurs="1" maxOccurs="1" />
  </choice>
  <attribute name="type" type="string" use="optional" />
  <attribute name="component" type="string" use="optional" />
  <attribute name="name" type="string" use="optional" />
  <attribute name="version" type="string" use="optional" />
</complexType>
<complexType name="BodyType">
  <choice minOccurs="1" maxOccurs="1">
    <element ref="sssrmap:Request" minOccurs="0" maxOccurs="1" />
    <element ref="sssrmap:Response" minOccurs="0" maxOccurs="1" />
    <any minOccurs="0" maxOccurs="1" namespace="##other" />
  </choice>
</complexType>
<complexType name="SecurityTokenType" mixed="true">
  <simpleContent>
    <extension base="string">
      <attribute name="type" type="string" use="optional" />
      <attribute name="name" type="string" use="optional" />
    </extension>
  </simpleContent>
```

```
</complexType>
<complexType name="SignatureType">
  <choice minOccurs="2" maxOccurs="3">
    <element ref="sssrmap:DigestValue" minOccurs="1" maxOccurs="1" />
    <element ref="sssrmap:SignatureValue" minOccurs="1" maxOccurs="1" />
    <element ref="sssrmap:SecurityToken" minOccurs="0" maxOccurs="1" />
  </choice>
</complexType>
<complexType name="DigestValueType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<complexType name="SignatureValueType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<complexType name="EncryptedDataType">
  <choice minOccurs="0" maxOccurs="1">
    <element ref="sssrmap:EncryptedKey" minOccurs="1" maxOccurs="1" />
    <element ref="sssrmap:CipherValue" minOccurs="1" maxOccurs="1" />
    <element ref="sssrmap:SecurityToken" minOccurs="1" maxOccurs="1" />
  </choice>
</complexType>
<complexType name="EncryptedKeyType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<complexType name="CipherValueType">
  <simpleContent>
    <extension base="string">
      <attribute name="method" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<complexType name="RequestType">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element ref="sssrmap:Object" minOccurs="0" maxOccurs="unbounded" />
    <element ref="sssrmap:Option" minOccurs="0" maxOccurs="unbounded" />
    <choice minOccurs="0" maxOccurs="1">
      <element ref="sssrmap:Get" minOccurs="1" maxOccurs="unbounded" />
      <element ref="sssrmap:Set" minOccurs="1" maxOccurs="unbounded" />
    </choice>
    <element ref="sssrmap:Where" minOccurs="0" maxOccurs="unbounded" />
    <element ref="sssrmap:Data" minOccurs="0" maxOccurs="unbounded" />
    <element ref="sssrmap:Count" minOccurs="0" maxOccurs="1" />
    <any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </choice>
  <attribute name="action" type="string" use="required" />
  <attribute name="actor" type="string" use="required" />
  <attribute name="id" type="string" use="optional" />
  <attribute name="chunking" type="boolean" use="optional" />
  <attribute name="chunkSize" type="positiveInteger" use="optional" />
</complexType>
<complexType name="ResponseType">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element ref="sssrmap:Status" minOccurs="1" maxOccurs="1" />
    <element ref="sssrmap:Count" minOccurs="0" maxOccurs="1" />
    <element ref="sssrmap:Data" minOccurs="0" maxOccurs="unbounded" />
    <element ref="sssrmap:File" minOccurs="0" maxOccurs="unbounded" />
    <any minOccurs="0" maxOccurs="unbounded" namespace="##other" />
  </choice>
  <attribute name="object" type="string" use="optional" />
  <attribute name="action" type="string" use="optional" />
  <attribute name="id" type="string" use="optional" />
  <attribute name="chunkNum" type="integer" use="optional" />
  <attribute name="chunkMax" type="integer" use="optional" />
</complexType>
<complexType name="ObjectType">
  <simpleContent>
    <extension base="string">
      <attribute name="join" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<complexType name="GetType">
  <attribute name="name" type="string" use="required" />
  <attribute name="object" type="string" use="optional" />
  <attribute name="op" type="sssrmap:GetOperatorType" use="optional" />
  <attribute name="units" type="string" use="optional" />
</complexType>
<simpleType name="GetOperatorType">
  <restriction base="string">
    <enumeration value="Sort" />
    <enumeration value="Tros" />
    <enumeration value="Count" />
    <enumeration value="Sum" />
    <enumeration value="Max" />
    <enumeration value="Min" />
    <enumeration value="Average" />
    <enumeration value="GroupBy" />
  </restriction>
</simpleType>
<complexType name="SetType">
  <simpleContent>
    <extension base="string">
      <attribute name="name" type="string" use="required" />
      <attribute name="op" type="sssrmap:SetOperatorType" use="optional" />
      <attribute name="units" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<simpleType name="SetOperatorType">
  <restriction base="string">
    <enumeration value="Assign" />
    <enumeration value="Inc" />
    <enumeration value="Dec" />
  </restriction>
</simpleType>
<complexType name="WhereType">
  <simpleContent>
    <extension base="string">
      <attribute name="name" type="string" use="required" />
      <attribute name="op" type="sssrmap:WhereOperatorType" use="optional" />
      <attribute name="conj" type="sssrmap:ConjunctionType" use="optional" />
      <attribute name="group" type="integer" use="optional" />
      <attribute name="units" type="string" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<simpleType name="WhereOperatorType">
  <restriction base="string">
    <enumeration value="EQ" />
    <enumeration value="GT" />
    <enumeration value="LT" />
    <enumeration value="GE" />
    <enumeration value="LE" />
    <enumeration value="NE" />
    <enumeration value="Match" />
  </restriction>
```

```
</simpleType>
<complexType name="OptionType">
  <simpleContent>
    <extension base="string">
      <attribute name="name" type="string" use="required" />
      <attribute name="op" type="sssrmap:OptionOperatorType" use="optional" />
      <attribute name="conj" type="sssrmap:ConjunctionType" use="optional" />
    </extension>
  </simpleContent>
</complexType>
<simpleType name="ConjunctionType">
  <restriction base="string">
    <enumeration value="And" />
    <enumeration value="Or" />
    <enumeration value="AndNot" />
    <enumeration value="OrNot" />
  </restriction>
</simpleType>
<simpleType name="OptionOperatorType">
  <restriction base="string">
    <enumeration value="Not" />
  </restriction>
</simpleType>
<complexType name="DataType">
  <sequence>
    <any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="optional" />
  <attribute name="type" type="sssrmap:Type" use="optional" />
</complexType>
<simpleType name="Type">
  <restriction base="string">
    <enumeration value="XML" />
    <enumeration value="Binary" />
    <enumeration value="String" />
    <enumeration value="Int" />
    <enumeration value="Text" />
    <enumeration value="HTML" />
  </restriction>
</simpleType>
<complexType name="FileType">
  <sequence>
    <any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="unbounded" />
  </sequence>
  <attribute name="name" type="string" use="optional" />
  <attribute name="owner" type="string" use="optional" />
```

```
    <attribute name="group" type="string" use="optional" />
    <attribute name="mode" type="string" use="optional" />
    <attribute name="compressed" type="boolean" use="optional" />
    <attribute name="encoded" type="boolean" use="optional" />
  </complexType>
  <complexType name="StatusType">
    <choice minOccurs="1" maxOccurs="unbounded">
      <element ref="sssrmap:Value" minOccurs="1" maxOccurs="1" />
      <element ref="sssrmap:Code" minOccurs="1" maxOccurs="1" />
      <element ref="sssrmap:Message" minOccurs="0" maxOccurs="1" />
      <any minOccurs="0" maxOccurs="unbounded" namespace="##other" />
    </choice>
  </complexType>
  <simpleType name="StatusValueType">
    <restriction base="string">
      <enumeration value="Success" />
      <enumeration value="Warning" />
      <enumeration value="Failure" />
    </restriction>
  </simpleType>
  <simpleType name="CodeType">
    <restriction base="string">
      <pattern value="[0-9]{3}" />
    </restriction>
  </simpleType>
</schema>
```

Node Object Schema

```
<?xml version="1.0" encoding="utf-8"?>
<schema
  targetNamespace="http://www.scidac.org/ScalableSystems/SSSRMAP"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:sssrmap="http://www.scidac.org/ScalableSystems/SSSRMAP"
  elementFormDefault="qualified">
  <?xml version="1.0" encoding="utf-8"?>
  <element name="Node" type="sssrmap:NodeType" />
  <element name="State" type="sssrmap:NodeStateType" />
  <element name="NodeState" type="sssrmap:NodeStateType" />
  <element name="Id" type="string" />
  <element name="NodeId" type="string"/>
  <element name="Name" type="string"/>
  <element name="Arch" type="string"/>
  <element name="OpSys" type="string"/>
  <element name="Description" type="string"/>
  <element name="Features" type="string"/>
```

```
<element name="Extension" type="sssrmap:ExtensionType"/>
<element name="Configured" type="sssrmap:ConsumableType" />
<element name="Utilized" type="sssrmap:ConsumableType" />
<element name="Available" type="sssrmap:ConsumableType" />
<element name="Processors" type="positiveInteger" />
<element name="Memory" type="sssrmap:MemoryType" />
<element name="Swap" type="sssrmap:MemoryType" />
<complexType name="NodeType">
  <choice minOccurs="0" maxOccurs="unbounded">
    <choice minOccurs="1" maxOccurs="2">
      <element ref="sssrmap:NodeState" minOccurs="0" maxOccurs="1" />
      <element ref="sssrmap:State" minOccurs="0" maxOccurs="1" />
    </choice>
    <choice minOccurs="1" maxOccurs="2">
      <element ref="sssrmap:NodeId" minOccurs="0" maxOccurs="1"/>
      <element ref="sssrmap:Id" minOccurs="0" maxOccurs="1" />
    </choice>
    <element ref="sssrmap:Name" minOccurs="0" maxOccurs="1"/>
    <element ref="sssrmap:Description" minOccurs="0" maxOccurs="1"/>
    <element ref="sssrmap:Arch" minOccurs="0" maxOccurs="1" />
    <element ref="sssrmap:OpSys" minOccurs="0" maxOccurs="1" />
    <element ref="sssrmap:Features" minOccurs="0" maxOccurs="1"/>
    <element ref="sssrmap:Extension" minOccurs="0" maxOccurs="1"/>
    <element ref="sssrmap:Configured" minOccurs="0" maxOccurs="1"/>
    <element ref="sssrmap:Utilized" minOccurs="0" maxOccurs="1"/>
    <element ref="sssrmap:Available" minOccurs="0" maxOccurs="1"/>
  </choice>
</complexType>
<complexType name="NodeStateType">
  <simpleContent>
    <extension base="sssrmap:NodeStateValue">
      <attribute name="lastSeen" type="string" />
    </extension>
  </simpleContent>
</complexType>
<simpleType name="NodeStateValue">
  <restriction base="string">
    <enumeration value="Up" />
    <enumeration value="Down" />
    <enumeration value="Invalid" />
  </restriction>
</simpleType>
<complexType name="ExtensionType">
  <simpleContent>
    <extension base="string">
      <attribute name="name" type="string" use="required" />
```

```
        <attribute name="type" type="string" use="optional" />
      </extension>
    </simpleContent>
  </complexType>
  <complexType name="ConsumableType">
    <choice minOccurs="0" maxOccurs="3">
      <element ref="sssrmap:Processors" minOccurs="0" maxOccurs="1" />
      <element ref="sssrmap:Memory" minOccurs="0" maxOccurs="1" />
      <element ref="sssrmap:Swap" minOccurs="0" maxOccurs="1" />
    </choice>
  </complexType>
  <complexType name="MemoryType">
    <simpleContent>
      <extension base="decimal">
        <attribute name="units" type="sssrmap:UnitsType" use="optional" />
        <attribute name="metric" type="sssrmap:MetricType" use="optional" />
      </extension>
    </simpleContent>
  </complexType>
  <simpleType name="UnitsType">
    <restriction base="string">
      <enumeration value="B"/>
      <enumeration value="KB"/>
      <enumeration value="MB"/>
      <enumeration value="GB"/>
      <enumeration value="TB"/>
      <enumeration value="PB"/>
      <enumeration value="EB"/>
      <enumeration value="ZB"/>
      <enumeration value="YB"/>
      <enumeration value="NB"/>
      <enumeration value="DB"/>
    </restriction>
  </simpleType>
  <simpleType name="MetricType">
    <restriction base="string">
      <enumeration value="Max"/>
      <enumeration value="Avg"/>
      <enumeration value="Min"/>
      <enumeration value="Tot"/>
    </restriction>
  </simpleType>
</schema>
```

APPENDIX D. Sample Infiniband Network XML

Host Channel Adapter

This appendix contains two sample `Node` elements in a query response from the Fountain server. In each example, the information was obtained from Fountain when it was compiled with support to monitor an InfiniBand network. The first example is from a single node with one Host Channel Adapter (HCA) and one port. The second example is from a large switch containing multiple smaller switch chips. Each smaller chip is displayed as a distinct `Device` element under the single `Node` element representing the switch. Certain elements in the second example have been omitted for brevity purposes, their information is essentially the same as what is shown in the first example with a single HCA. In the first example the `NodeId` element is the HCA's GUID, while in the second example it is the switch's system image GUID.

```
<Node>
  <NodeId>0002c90200003448</NodeId>
  <Arch>Infiniband</Arch>
  <Network type="Infiniband">
  <Device>
    <ID>0002c90200003448</ID>
    <Vendor>Redswitch</Vendor>
    <Lid>35</Lid>
    <Description>MT23108 InfiniHost Mellanox Technologies</Description>
    <Type>HCA</Type>
    <Ports>
    <PortCount>2</PortCount>
    <Port>
      <Number>1</Number>
      <RemoteDevice port="2">0002c90109fb36b8</RemoteDevice>
      <SendBytes units="bytes">648</SendBytes>
      <ReceiveBytes units="bytes">576</ReceiveBytes>
      <SendRate>
        <Bytes>39.865</Bytes>
        <Packets>0.554</Packets>
      </SendRate>
      <ReceiveRate>
        <Bytes>39.865</Bytes>
        <Packets>0.554</Packets>
      </ReceiveRate>
      <SymbolErrors>60</SymbolErrors>
      <Counters>true</Counters>
      <LastSeen>Mon Jun 5 15:09:38 2006</LastSeen>
      <Width>4X</Width>
      <Speed units="Gigabits/sec">2.5</Speed>
    </Port>
    </Ports>
  </Device>
  </Network>
</Node>
```

Switch

```
<Node>
  <NodeId>0002c9010a038090</NodeId>
  <Arch>Infiniband</Arch>
  <Network type="Infiniband">
    <Device>
      <ID>0002c90109faf120</ID>
      <Type>Switch</Type>
      <Ports>
        <PortCount>8</PortCount>
        ...
      </Ports>
      ...
    </Device>
    <Device>
      <ID>0002c90109faf128</ID>
      <Type>Switch</Type>
      <Ports>
        <PortCount>8</PortCount>
        ...
      </Ports>
      ...
    </Device>
    ...
  </Network>
</Node>
```

BIBLIOGRAPHY

[1] P. Marginean A. Alexandrescu. Change the way you write exception-safe code. http://www.ddj.com/dept/cpp/184403758, 2003. [Online; accessed April 30, 2006].

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[3] A. Alexandrescu. Assertions. http://www.ddj.com/dept/cpp/184403861, 2003. [Online; accessed April 30, 2006].

[4] A. Alexandrescu. Enforcements. http://www.ddj.com/dept/cpp/184403864, 2003. [Online; accessed April 30, 2006].

[5] Apache http server documentation. http://httpd.apache.org/docs/trunk/, 2006. [Online; accessed May 22, 2006].

[6] The standard librarian: Sorting in the standard library. http://www.ddj.com/dept/cpp/184403792, 2000. [Online; accessed June 13, 2006].

[7] R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs, 2001.

[8] T. Tromey G. V. Vaughan, B. Elliston and I. L. Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders, 2000.

[9] M. Huang. A performance comparison of tree and ring topologies in distributed systems. Master's thesis, Iowa State University, 2004.

[10] 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, April 2005.

[11] P. Wyckoff J. Wu and D. K. Panda. Pvfs over infiniband: Design and performance evaluation. In *Proc. IEEE International Parallel Processing*, pages 125–132, Kaohsiung, Taiwan, April 2003.

[12] D. Jackson. Maui cluster scheduler. `http://www.supercluster.org/projects/maui/index.shtml`, 2005. [Online; accessed May 31, 2006].

[13] M. Jette and M. Grondona. Slurm: Simple linux utility for resource management, 2002.

[14] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, 1999.

[15] Loadleveler: Get the most out of your computing resources. `http://www.ibm.com/servers/eserver/clusters/software/loadleveler.html`, 2006. [Online; accessed May 2, 2006].

[16] Load sharing facility. `http://www.platform.com`, 2006. [Online; accessed May 2, 2006].

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

[18] O. Matlin, E. Lusk, and W. McCune. Spinning parallel systems software, 2002.

[19] The apache xml project. `http://xml.apache.org/xerces-c`, 2006. [Online; accessed April 30, 2006].

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

[21] Open infiniband alliance. `http://www.openib.org`, 2005. [Online; accessed November 10, 2005].

[22] Portable batch system. http://www.openpbs.org, 2006. [Online; accessed May 2, 2006].

[23] R.S. Studham 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, September 2004.

[24] T. C. Ruys. Spin tutorial: How to become a spin doctor. In *Proc. International SPIN Workshop, volume 2318 of LNCS*, pages 6–14, Grenoble, France, September 2002.

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

[26] Scinet: The world's fastest network. http://sc05.supercomputing.org/exhibits/scinet.php, 2006. [Online; accessed June 6, 2006].

[27] 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, September 2002.

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

[29] W. R. Stevens. *UNIX Network Programming.* Prentice Hall, 1998.

[30] tcpstat: network interface statistical monitoring. http://www.frenchfries.net/paul/tcpstat/, 2006. [Online; accessed April 10, 2006].

[31] Top500 supercomputing sites. http://www.top500.org, 2005. [Online; accessed June 16, 2006].

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

[33] Cray xt3 supercomputer. http://www.cray.com/products/xt3, 2006. [Online; accessed June 2, 2006].

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting this research and writing this thesis. First and foremost, Dr. Brett Bode for his guidance, and support throughout this research and the writing of this thesis. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Srinivas Aluru and Dr. Robyn Lutz. Lastly, I would like to thank all of my fellow students and colleagues in the Scalable Computing Laboratory. Without their support, this work would not have been possible.

The United States government has assigned DOE Report number IS-T 2613 to this thesis. Notice: This document has been authored by the Iowa State University of Science and Technology under Contract No. W-7405-ENG-82 with the U.S. Department of Energy. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wde license to publish or reproduce the published form of this document, or allow others to do so, for U.S. Government purposes.