LIGHT-WEIGHT HIERARCHICAL CLUSTERING MIDDLEWARE FOR PUBLIC-RESOURCE COMPUTING

By

AUSTIN ROYCE GILBERT

Bachelor of Science

Oklahoma State University

Tulsa, Oklahoma

2002

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2007

LIGHT-WEIGHT HIERARCHICAL CLUSTERING

MIDDLEWARE FOR PUBLIC-RESOURCE

COMPUTING

Thesis Approved:

Dr. István Jónyer
Thesis Advisor

Dr. Johnson Thomas

Dr. XiaoLin Li

Dr. A. Gordon Emslie
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank Professor Jónyer for graciously agreeing to overseeing my thesis work. I would also like to express my gratitude to Professors Thomas and Li for all of their wonderful insights and for sitting on my advisory board.

I would like to thank the Department of Computer Science for supporting me during my course work with a teacher's assistantship. The experience I gained was invaluable.

A special thanks to Earl Goodman, Jr. and all of my good friends at Tulsa Community College for offering me work and friendship, and for availing their resources to me over the course of my thesis work. I appreciate it immensely.

Finally, I would like to thank my darling wife Maryna for all her support and encouragement along the way. I never would have made it without you.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

There are many areas of science where conducting meaningful research requires large amounts of computation. In many cases, research activities are restricted due to limited availability of computational resources. These limitations are especially apparent at smaller institutions with limited research budgets (Chambers and Poore, 1975; Scragg, 1987; Chavey, 1995; Best, Chamberlain, Maene, 2002), and for projects with very large computational needs (Seti@Home). Public-resource computing (Sullivan, *et al.*, 1997; Anderson *et al.*, 2002; Loewe, 2002; Anderson, 2003; Anderson, 2004), which builds on earlier works in computer-resource sharing (Shoch and Hupp, 1982; Nichols, 1987; Litzkow, 1987; Litzkow, 1988; Theimer and Lantz, 1988), offers a solution for expanding access to resources beyond those dedicated to research. However, the Internet bandwidth required to host large-scale projects (SETI, 2004) may represent another barrier (Gammill, 1990). Moving the public-resource computing paradigm to a hierarchically scalable architecture is expected to be the path to overcoming the bandwidth barrier, thereby improving resource accessibility. This research explores such an approach in a lightweight hierarchical clustering middleware.

Today's most prominent public-resource computing system, the Berkeley Open Infrastructure for Network Computing (BOINC) (Anderson, 2003; Anderson, 2004), utilizes a centrally distributed client-server architecture, the scalability of which is directly proportional to the bandwidth available at the central server and/or the server's performance (Ibe, Choi, and Trivedi, 1993; Nelno, *et al.*, 1995; Arlitt, Krishnamurthy, and Rolia, 2001). We are proposing a public-resource computing middleware utilizing a hierarchically distributed architecture as an attempt improve scalability. The proposed middleware will employ clusters of clusters, grouping resources according to Internet addresses in an attempt to conform optimally to the underlying network's topology. Some of the main ideas behind our research are as follows.

We expect that organizing participants into clusters will streamline work requests and the return of results. Clustering will reduce the number of clients interacting with a single server, allowing the system to take advantage of low latency, high-bandwidth connections for distributing/collecting work. We expect Local Area Network (LAN) (Metcalfe and Boggs, 1976; DEC, 1980; Shoch, 1980; Bux, 1981; Rosenthal, 1982; Graube, 1982; IEEE, 1983) and Wide Area Network (WAN) (Bell, 1986; Haas and Cheriton, 1987) connections to constitute a significant portion of smaller clusters within the system.

We expect that distributing work over the highest bandwidth lowest latency connections available will reduce the overall system latency. When these connections are LAN/WAN connections they also help to regulate Internet bandwidth utilization on the client side.

We expect that reducing the number of clients interacting with a single server will reduce the potential for concurrent client interactions and will simplify the communication-scheduling problem. The potential for better scheduling could further improve network efficiency by enabling better coordination of network resource utilization. For the purposes of this thesis, we will leave investigations into impact of clustering on the communication-scheduling problem as future work.

The motivations for this project are as follows. We expect to enable organizations to make better use of their existing computational infrastructure by harnessing the idle CPU cycles of their computers. We expect to lower the amount of bandwidth needed for hosting a public-resource computing project by improving network utilization. We expect that lowering bandwidth demands will make the computational model accessible to a wider range of institutions that could not otherwise afford to host public-resource computing projects. Finally, we expect the improved availability of computational resources offered by the public-resource computing model will encourage the model’s use in new and interesting work.

CHAPTER 2

LITERATURE REVIEW

Even though the term *public-resource computing* has not be formally defined, it has been widely used in the literature in such contexts that imply the following two constraints. First, because access to resources is being donated by third party volunteers, the middleware must only use a resource's idle computing time. Restricting the system to idle computing cycles is something of a tradition (Shoch and Hupp, 1982; Theimer, Lantz, and Cheriton, 1985; Theimer, 1986; Litzkow, 1987; Theimer and Lantz, 1988; Freeley, *et al.*, 1991). This practice is sometimes called *cycle stealing* (Litzkow, 1987; Litzkow, 1988; Tannenbaum and Litzkow, 1995; Bhatt, *et al.*, 1997; Rosenberg, 2002). Second, the middleware must not require administrative privileges or special network configurations to run. This constraint assures maximum accessibility to the system, and allows the inclusion of resources volunteered by users who either don't have the skill or the privileges to modify their network configuration. We feel meeting this constraint necessitates the use of client-driven communications, that is, all communication between computing resources and project hosting servers must be initiated by the client resources. Servers are unable to initiate communication directly to the clients. The majority of distributed computing systems fail to meet one or both of these constraints, making them unsuitable for use as a public-resource computing middleware.

2.1 Distributed Virtual Parallel Machines

Distributed Virtual Parallel Machines (DVPMs) are distributed computing systems designed to emulate a massively parallel computer using inter-networked commodity computers. There has been no shortage of offerings in this area. Examples include Parallel Virtual Machine (PVM) (Sunderam, 1990; PVM, 2005), Message Passing Interface (MPI) (Forum, 1995; MPI, 2005), Heterogeneous Adaptable Reconfigurable NEtworked SystemS (HARNESS) (Beck, *et al.*, 1999), and Visper (Stankovic and Zhang, 2002) to name a few.

MPI and PVM are not suitable for use in public-resource computing systems for a few reasons. Most notably they lack task distribution mechanisms. Programs must be installed on the participating machines, be available over a shared network drive, such as NFS, or must be uploaded via a remote shell prior to use. Access to shared network drives cannot be assumed in a public-resource computing environment. Uploading programs over a remote shell would require inbound network access to participating resources, violating our second constraint.

Visper is built on top of the Aglets software agent platform (Aglets, 2002). Aglets utilizes an asynchronous messaging system which relies on inbound network communications, making it unsuitable for use in public-resource computing.

Heterogeneous Adaptable Reconfigurable NEtworked SystemS (HARNESS) is a metacomputing infrastructure for deploying fault-tolerant DVPMs and running fault-tolerant MPI (Fagg, Bukovsky, and Dongarra, 2001). HARNESS utilizes a peer-to-peer distributed control algorithm to remove the single point of failure found in other DVPMs. However, this algorithm necessitates the need for inbound communication making it unsuitable for use in a public-resource computing system.

2.2 Metacomputing Approaches

IceT (Gray and Sunderam, 1999) is a metacomputing system written in Java. IceT gains access to remote resources via HTTP requests. The necessity of running an HTTP server on client resources negates its use in a public-resource computing setup.

Condor (Litzkow, 1988) is a cycle-stealing metacomputing system. Condor uses a client-server architecture where servers schedule jobs on “pools” of idle clients after locating them on their network. The server-driven communications for job scheduling make Condor unsuitable for use in public-resource computing systems. In the last few years Condor has been integrated into the Globus Toolkit (Foster and Kesselman, 1997) and released as Condor-G (Frey, *et al.*, 2002).

XGrid (Apple, 2005) is a distributed computing middleware focused on ease of use and configuration. It features automatic resource discovery on LANs through mDNS (Cheshire, 2005). XGrid has been configured for use in a public-resource computing system (Parnot, 2005), though it is not well suited for this role for at least two reasons. First, XGrid is platform specific. Projects can only be hosted on Apple Macintosh computers. Although XGrid use BEEP (Rose, 2001), an open standard for communications protocols that allows for the development of third party clients, to date only Unix-like operating systems have been supported (Côté, 2004). The Unix XGrid agents can only participate as processing clients (i.e., no job submission). Second, if an executable used by a task is not installed locally on the client's machine, the executable is copied to the client. When the task is completed the executable is removed. No file or executable caching scheme is employed. This approach exacerbates the current problems public-resource computing systems have with bandwidth utilization.

2.3 Web-browser and Screen-saver Distributed Systems

Javelin (Christiansen, *et al.*, 1997) is a web-browser driven distributed computing system. Javelin is client-driven and can be used for public-resource computing, however, it has some severe limitations. Applets are not allowed to access the local filesystem, therefore calculated results must reside in memory. Also, the tasks can only be defined in the Java language. Our middleware will support the execution of tasks written in a wide range of scripting and programming languages.

2.4 Peer-to-Peer Metacomputing and Grid Computing Approaches

OurGrid (Andrade *et al.*, 2003) is a peer-to-peer computational middleware. OurGrid's use of a peer-to-peer architecture necessitates that *all* peers be able to receive inbound network communication. This violates our second constraint for use as a public-resource computing system.

The Globus Metacomputing Toolkit (Foster and Kesselman, 1997) is a toolkit for building metacomputing infrastructures, which is more recently termed *grid computing*. (Foster and Kesselman, 1999) Globus requires inbound and outbound communication which makes it unsuitable for use in a public-resource computing system.

The gLite framework (GLite, 2005) is a metacomputing approach based on the AliEn grid framework (AliEn, 2007). GLite's architecture closely resembles Globus. The gLite framework has been chosen as the base for analyzing data from the ALICE project (ALICE, 2007), a CERN Large Hadron Collider project (LHC, 2007). For our purposes it has shortcomings similar to the Globus framework.

OCEAN (Padala, *et al.*, 2003) is a peer-to-peer metacomputing framework with a heavy emphasis on utilizing a computational economy (Buyya, Abramson, and Giddy, 2001; Buyya and Vazhkudal, 2001) approach to distributed computing and resource sharing. The peer-to-peer nature of OCEAN's resource matching algorithm makes it unsuitable for use in a public-resource computing system, because the peer-to-peer architecture necessitates the need for inbound network communications which violates our second constraint for public-resource computing systems.

2.5 Public-Resource Computing Middleware

Grid MP. United Devices' Grid MP middleware (United Devices, 2005; GridMP, 2005) is a public-resource computing middleware built around a computational economy model (Buyya, Abramson, and Giddy, 2001; Buyya and Vazhkudal, 2001). Grid MP is a commercial product and must be licensed before deployment in public-resource computing projects. The commercial nature of the middleware also means little technical information is available regarding its architecture, preventing it from being highly available and making it difficult to assess fairly.

ZetaGrid (Zeta, 2002) is cycle-stealing middleware developed in Java and funded by IBM. It features encrypted communications and nonrepudiation of client and server exchanges via digital signatures. ZetaGrid has a client-server architecture where work flow is demand driven by client availability. One of the more interesting features of ZetaGrid is the incorporated trust model, which is used to determine the amount of work a client is entrusted with and how frequently a client's results are reviewed for errors. Unfortunately, at the time of this writing, ZetaGrid's trust algorithms used are not available in the literature. Its security features make ZetaGrid an attractive option for public-resource computing project distributing sensitive data to trusted clients over the untrusted Internet.

Berkeley Open Infrastructure for Network Computing (BOINC) (Anderson, 2003; Anderson, 2004) is a cycle-stealing middleware designed for deployment in public-resource computing environments. BOINC is a mature and well developed middleware, but lacks sophistication in bandwidth utilization and ease of use.

Achieving high performance for large projects with BOINC can require large quantities of bandwidth to serve the large number of participating computing resources. For example, in 2004, the SETI@home project, which utilizes the BOINC architecture, had to throttle back the computation rate of their distributed application in order to limit their bandwidth consumption to 30 Mbits/s (SETI, 2004). A full T-1 connection has a maximum rate of 1.536 Mbits/s, meaning that even when throttled down, the SETI project was using approximately 19.5 T-1 lines. This example highlights some of the issues with hosting larger projects.

BOINC does not take advantage of the underlying network topology. No consideration is given to proximity of participants. For example, BOINC clients residing on the same LAN work independently to retrieve work and return results. Coordination of these clients is possible and we expect coordination to lead to a better communication model.

Installing, configuring, and running a project on BOINC is far from trivial. Several services must be installed and configured, including the Apache HTTP server (Apache, 2005), the MySQL relational database (MySQL, 2005), and the Python scripting language (Python, 2005). Each additional service adds work for systems administrators, and requires considerable expertise. Additionally, each network service introduces security concerns.

A major goal of our project is to simplify the effort needed to host public-resource computing projects by reducing the amount of configuration involved. A second project goal is to minimize security threats by requiring as few new network services as possible.

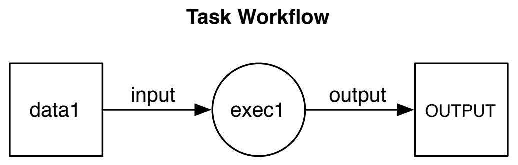
CHAPTER 3

SimpleDS: A LIGHTWEIGHT HIERARCHICAL CLUSTERING MIDDLEWARE

In this chapter we describe our implementation of a hierarchical clustering middleware named SimpleDS in detail. The outline of this chapter is as follows. Section 3.1 provides a high-level overview of work-flow in our system. Section 3.2 introduces the system architecture. Finally, section 3.3 presents proposed clustering mechanisms for SimpleDS.

3.1 Work Flow

The largest unit of work in SimpleDS is the *project.* A project is a group of related *tasks, task-files*, and *results*. A task defines some work to be done by the system. Tasks are described by *task manifests*. A task manifest describes the task-files needed to complete a task. Task-files include data-files and executables. A task manifest (fig. 3.1) is a string describing the relationship between a task's executables and the data files, that is, which data files are inputs to which executables. Further, manifests describe output files resulting from running the defined executable(s). The last output files produced by the completion of a task are the task's *results.* Tasks are distributed throughout the system for processing and results are collected. Results are data files describing the outcome of a task. Typically, result files will be ASCII (ASCII, 1986) text files. Binary data files can be used but caution must be exercised to ensure cross-platform compatibility.



Related Task Manifest

```
data:( data1 )
exec:( exec1 ):( Darwin-8.0.0-powerpc )
{exec1}:( data1 ):( OUTPUT )
```

Figure 3.1. Task Manifest and Workflow Diagram

3.2 System Architecture

SimpleDS uses a hierarchically distributed architecture to facilitate distributed computation, as shown in figure 3.2. A hierarchically distributed architecture consists of three components: a *central server*, *intermediate servers*, and *clients*. In SimpleDS, central severs and intermediate servers are called *cluster managers.* Cluster managers are responsible for hosting and distributing tasks, data files, and executable files. Cluster managers are also responsible for collecting results. Special cluster managers, called *root nodes*, provide the functionality for task submission/creation. Results collected by cluster managers are forwarded to the root nodes where the associated tasks were created.

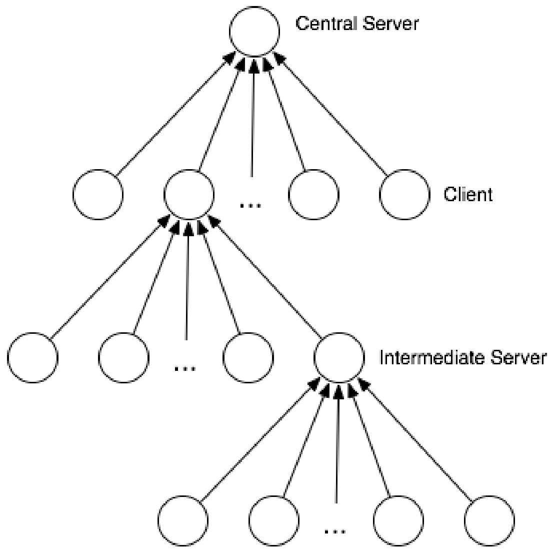


Figure 3.2. Hierarchically Distributed Architecture

A cluster manager and its associated nodes are referred to as a *cluster*. We wish clusters in SimpleDS to be dynamic, forming automatically when it is advantageous to the overall system performance and dissolving when they are no longer needed. The performance metrics used to determine overall system performance have yet to be determine, initially both turnaround time and throughput will be measured. We refer to the process of cluster formation and disbandment as *clustering*. Many clustering algorithms are possible. We feel that clusters will be most useful when they form along the boundaries of the underlying network topology. We base this belief on the principle of network locality (Lorence and Satyanarayanan, 1990; Freedman and Mazieres, 2003; Pias, *et al.*, 2003). Network locality is also referred to as *network proximity* (Castro, *et al.*, 2002; Amini and Schulzrinne, 2004; Zhan, 2004). It is sometimes discussed in conjunction with the related topics of *topology-awareness* (Ratnasmay, 2002; Karonis, *et al.*, 2002; Castro, *et al.*, 2003), *network-awareness* (Krishnamurthy and Wang, 2000; Massoulie, Kermarrec, and Ganesh, 2003), or *locality-awareness* (Zhang, *et al.*, 2004). Network locality simply says that the closer two nodes are together on the physical network, the fewer routing hops between them, the lower the communication latency will be. It should be noted that this principle is a generality and does not hold true for every possible case. Network locality is typically generalized into a distance metric (Francis, *et al.*, 2001; Huffaker, *et al.*, 2002; Ng and Zhang, 2002; Amini and Schulzrinne, 2004; Costa, *et al.*, 2004; Cox, *et al.,* 2004), many of which are based on Roundtrip Time (RTT). Haffaker et al. (Haffaker, 2002) have demonstrated that the 24-hour moving average minimum RTT is generally the best estimator for network locality.

3.3 Clustering

The purpose of clustering is twofold. From the project hosting perspective, we aim to share the burden of hosting with amicable peers. From the client/participant perspective we seek to coordinate the use of shared network infrastructure, both to prevent overuse and to exploit lower latency connections available across a shared infrastructure. In both cases, we seek to conform to the underlying network topology.

Private addressing (Rekhter, *et al.*, 1996) introduced a dichotomy into the Internet. There are host addresses visible from the Internet and addresses only visible by nodes residing on the same LAN. We feel this dichotomy warrants the use of two separate clustering algorithms and work discovery mechanisms - one to take advantage of shared local infrastructures (LANs and WANs), and one for Internet level clustering.

3.3.1 LAN Clustering

Our LAN clustering algorithm is client driven. Clients are either compute nodes or cluster managers needing work. As clients become available to perform work they seek a local cluster manager. Resource discovery occurs over IP multicast channels (Deering, 1989). The link-local layer is searched first, followed by the organization-local layer. A client seeking work broadcasts a work request to all participants. Local cluster managers respond with the amount of work they have and the highest user assigned priority for the work. The client employs a lottery-scheduling algorithm (Waldspurger and Weihl,

1994) to choose from the list of cluster manager responses. Cluster managers with the most work at the highest priority will be given the most lottery tickets. A random number is chosen to determine the lottery winner. The client then associates itself with the lottery winner. The client will stay associated with the cluster manager as long as it can supply a constant flow of work. When work shortages arise, the client will restart the browsing process. If no work is available locally, the cluster manager will begin the Internet-level browsing process. Any work retrieved from an Internet cluster will be shared among the local clients. The goal is to have as few cluster managers downloading work from the Internet as possible, preferring a steady lower bandwidth stream to short high bandwidth bursts. In this way, we can avoid the scenario where thousands of nodes saturate their local Internet connection downloading work from Internet servers.

3.3.2 Internet Clustering

The Internet clustering algorithm is also client driven. When clients are available to process tasks, or cluster managers need more work, they seek an Internet cluster manager with work to distribute. Being an Internet cluster manager requires an Internet routable address. In an environment with Network Address Translation (NAT) (Egevang and Francis, 1994) port redirection must be configured, which requires administrative privileges and know-how, hence participants are never required to participate as Internet cluster managers. The search for a cluster manager begins at the LAN/WAN level. If no local cluster managers are discovered then the client continues the search for work with a web service request for a list of available projects. The function of the list is to serve as a global directory of projects. Projects hosted on root nodes with the ability to support Internet clusters will automatically register themselves with the project directory web service. In the future, the web service will utilize a highly available redundant architecture supporting data replication. The back-end will likely be implemented using a Lightweight Directory Access Protocol (LDAP) (Wahl, Howes, and Kille, 1997) directory, or a similar directory structure. Initially the list provided by the web service will contain each project's name, description, contact, and the IP address for its root node. In the future, more information may be provided. Nodes will examine the list of IP addresses and perform the longest prefix matching algorithm (LPM) (Fuller, *et al.*, 1993; Ruiz-Sanchez, Biersack, and Dabbous, 2001) on the pre-CIDR addresses (Fuller, *et al.*, 1993). If no exact match is given, the node will select the IP which is closest to its own, where 203.0.5.2 would be a match to 200.10.45.7 when compared with 210.2.4.5. Three to five candidate cluster managers will be selected. The node will then test its roundtrip time with each candidate and select the node with the lowest result. The node will then contact the candidate and request to join its cluster. The cluster manager then has the option to accept the request, reject the request, or refer the requesting node to another cluster. If the request is rejected, the seeking node will move on to the cluster manager with the next lowest roundtrip time. If the request is accepted, the node will enter its work processing phase by requesting work. If a cluster manager has several other cluster managers associated under it, it may elect to refer the requesting node to one of them. In this case, the referring cluster manager returns a list of its associated cluster managers to the requesting node. The node would then restart the browsing algorithm with this list.

The purpose of the referral process is to direct incoming nodes to cluster managers with the closest network proximity in order to establish and maintain a structure that best conforms to the underlying network topology.

Once a cluster manager is no longer providing an adequate amount of work, a node may restart the browsing process to maintain a constant flow of work. Hence, nodes swarm to new work sources when old work sources can no longer supply an adequate amount of work. In this way, the clustering algorithm is dynamic. Clusters form around work sources, aligning themselves to the underlying topology, the clusters cease growing once an adequate work force has been assembled and they disband as the work source runs low.

We will investigate the merit of this approach and make refinements as needed. We are already aware of several potential issues. One of which is the convoy effect (Vogels, van Renesse, Birman, 2003) exhibited by some distributed applications. At this point we are uncertain as to the impact the convoy effect will have, but we feel it is likely that an optimal or near optimal limit for the nesting of clusters must be found and adhered to.

CHAPTER 4

IMPLEMENTATION

In this chapter we will describe the system implementation. Our main design goals for the implementation were to make the system stable, robust (resistant to failure), flexible, and portable. The middleware is implemented in C++.

4.1 Implementation Overview

The system design process involved a certain amount of trial and error until the final design emerged which was both stable and robust. Our initial implementation was driven by a single complex state machine running in a single thread of execution. It proved neither robust nor stable. We learned from the design mistakes we made and moved to three threads and three simpler state machines. The result was a simpler system, but was still not robust or stable. It took a third redesign before the result yielded a system stable and robust enough to serve as a base for further development. We describe the final design in this chapter.

Our approach for the final redesign was to break the system functionality into independent modules, giving each module its own state machine and isolating each state machine in its own thread. For the majority of the modules, this approach allowed us to simplify the state machines down to three basic states: an action state, a sleep state, and a checkup state. Each module runs independently. We found this the fastest path toward a stable and robust system. When modules need to communicate, interaction occurs through table entries in the back-end database. One module may write information in the database and another reads/updates/deletes it at a later point in time. All database interactions occur in ACID transactions (atomic, consistent, isolated, durable) to assure a consistent state, even in the event of a power failure on the hosting computer. Further, our experience with our first implementation led us to design the implementation using the *crash-only* paradigm (Candea and Fox, 2003). During system start up each module has no assumptions about its state. They all begin in their checkup state to determine if they need to progress to their action state or their sleep state. Because they are independent, no module makes any assumptions about the state of other modules. Any module can fail without causing a complete failure of the middleware as a whole, enabling graceful degradation in the event of failure.

4.2 Development Environment

A middleware of this nature is a large undertaking. Utilizing existing libraries helped make the task marginally less difficult, but care had to be taken to ensure that each library incorporated into the project was stable and portable. We checked each library for portability, requiring support for FreeBSD, Macintosh OS X, Microsoft Windows XP, and Linux at the very least. For UNIX platforms, we required the library to compile cleanly with GCC (GCC, 2007) versions 3.4 and 4.0. On windows, we required native support for Microsoft Visual C++ (Microsoft, 2007) versions 7.0 (Visual Studio 2003) and 8.0 (Visual Studio 2005). VC++ 6.0 is not supported because of its lack of support for IPv4/IPv6 agnostic function calls. We also required libraries to support a variety of processor architectures, including PowerPC, Intel i386, and AMD64. The library vetting process was informal. This resulted in many delays due to necessary experimentation and testing of each library. The end gain, however, is a system that compiles and runs on Microsoft Windows XP and many varieties of UNIX without modification. Further, the system is postured to be easily, but not trivially, ported to other unsupported platforms.

We utilized the Boost Library collection (BOOST, 2003) to abstract idiosyncrasies between incompatible computing models, for example, the POSIX threading model versus Windows threading. In particular, we used the Boost Filesystem and Thread libraries throughout our middleware. Our middleware routinely handles file creation, deletion, and I/O. Interacting with files requires system dependent file paths. The Boost Filesystem library provides a useful file path abstraction which greatly simplified the task of creating portable filesystem code. Likewise, the Boost Thread library hides the differences in the POSIX thread model versus the Windows thread model, allowing a single consistent API for thread management, greatly improving the readability and maintainability of the code base.

Our middleware has to track a lot of information about tasks. We elected to use a pre-existing data store rather than develop our own proprietary solution. We initially considered implementing our data store using BerkeleyDB (Olson, Bostic, and Seltzer, 1999) as it is very portable and widely available. However, we elected to go with SQLite (SQLite, 2005) instead. SQLite is an embedded SQL database offering good performance. We chose SQLite for two reasons. First, SQLite has great language support, offering APIs in several programming and scripting languages, meaning that we could in turn offer our APIs in several programming and scripting languages. Secondly, we found a very useable and intuitive C++ API for SQLite called CppSQLite (Groves, 2004), which greatly eased the learning curve needed to use SQLite.

4.3 System Modules

The system is composed of seven modules and three support frameworks. The system modules are: the *Upload Manager*, the *Download Manager*, the *Multicast Manager*, the *Client Manager*, the *Cluster Browser*, the *Work Manager*, and the *Work Processor(s)*. The support frameworks are: the *Task Scheduler*, the *File Manager*, and the *Statistical Engine*.

4.3.1 The Work Processor

Each node may have zero or more work processors. Each work processor is isolated in its own thread. The default number of work processors is determined by the number of CPUs a system has. System specific calls are used to determine the number of CPUs present, e.g. *sysconf, sysctl,* and *GetSystemInfo*. The sole purpose of the work processor is to execute tasks. The *work processor* lifecycle (fig. 4.1) has four states: INIT, IDLE, CHECK_FOR_WORK, and PROCESS. The work processor starts in the INIT state where its local variables are initialed and immediately continues on into the CHECK_FOR_WORK state.

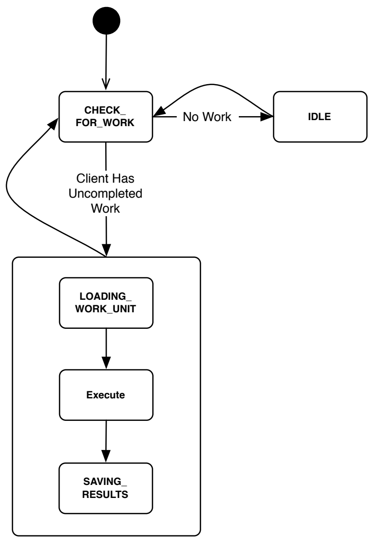


Figure 4.1. State Machine Diagram of Work Processor Thread

In the CHECK_FOR_WORK state, the *work processor* asks the *task scheduler* for a single task. If the *task scheduler* assigns a task, the *work processor* continues into the PROCESS state, otherwise it returns to the IDLE state.

In the IDLE state, the *work processor* thread sleeps. This prevents unnecessary use of the CPU when there are no tasks available for execution. The default sleep time is randomly selected between one second and two seconds.

In the PROCESS state, the *work processor,* sets up and executes assigned tasks. The PROCESS state is broken into three sub-states representing possible failure points: work unit loading, execution, and result saving. The temporary work directory is created during the loading phase. The required data files and executables are unpacked into the working directory. A process is forked and the process priority is changed to *idle* - on UNIX systems this is a *nice* level of 20, on Windows this is the IDLE_PRIORITY_CLASS. If the process priority cannot be changed to an acceptable level, the task exits with an error, otherwise the task is executed according to the manifest definition. Any error messages printed to STDERR by the executables are redirected to a file called ERRORS. Output printed to STDOUT are redirected to a file called OUTPUT. After the task execution completes, the executables and input data files are deleted and any remaining files are archived for return to the cluster manager. Finally, the temporary work directory is deleted from the host system. If any of these sub-states fail, the work processor sets the task status to TASK_ERROR, otherwise the task status is set to TASK_COMPLETE. The presence of the TASK_COMPLETE status does not indicate that a task was completed successfully, it merely indicates that the middleware was able to execute the tasks defined by the task manifest and was able to save the results. The ERRORS file must be examined to determine if the task generated runtime errors or not.

There are a couple of known shortcomings with our current implementation of the *work processor.* There are no middleware imposed limits on memory utilization, execution time, or CPU utilization of the task being processed. By CPU utilization we mean the percentage of the CPU being utilized. This is a metric the SETI@home project has shown to be a concern for volunteers due to temperature and power consumption issues (SETI, 2005b). Of course, on UNIX systems, our processes would be limited by the *ulimit* (IEEE, 2001) settings, but on Windows and some other platforms, we would have the ability to adversely affect the performance of the host system, *e.g.* exhausting available physical and virtual memory. These short comings need to be addressed in future releases.

4.3.2 The Work Manager

The *work manager* is the keystone of work flow in the system. It handles communications associated with task requests and determines the amount of work a node needs. It has four states: INIT, CHECK_FOR_WORK, REQUEST_WORK, and IDLE (fig. 4.2).

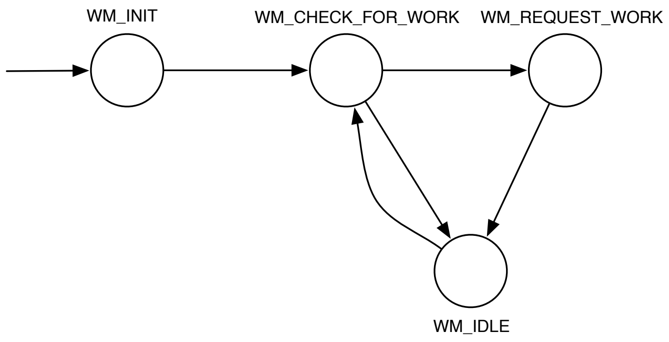


Figure 4.2. Work Manager State Machine Diagram

In SimpleDS, nodes may request multiple tasks simultaneously and the *work manager* maintains the queue of assigned tasks. All tasks and related files are kept on hand until they can be processed locally, reassigned to other nodes, or the tasks are overdue. The work queue increases the efficiency of the system by allowing the *work processors* to execute task after task without delays for network communication. The goal of the *work manager* is to maintain enough work on hand to ensure that the *work processors* are constantly busy. To this end, the *work manager* must decide when to make work requests and how much work to request. This is not an easy task. Having too much work on hand could mean not completing all tasks in a timely manner, and too little work means cycles wasted idling.

First, tasks in the system are not assumed to be homogeneous. The same executable with the same input may have different running times, even on similar hardware, depending on the class of application to which the task belongs. Further, no assumption can be made about the processing speed of clients. Benchmarks can estimate a client's processing speed, but cannot account for variability in the client's availability which affects the observed processing speed of the client. Since the system is volunteer based, each client's availability is expected to vary in unexpected ways, causing the observed processing speed to vary unexpectedly. Finally, no assumption is made about the mix of tasks assigned. A node may be assigned dozens or hundreds of tasks with durations averaging six hours, and then suddenly be assigned tasks with durations averaging 30 minutes. This makes the problem of determining how much work to keep on hand very challenging. However, this problem is not unlike problems faced in many engineering disciplines where a single variable needs to be regulated within a certain constraint while it is continually being affected by outside forces. One solution to such problems is a *controller*. A simple example of a controller is a thermostat which regulates room temperature.

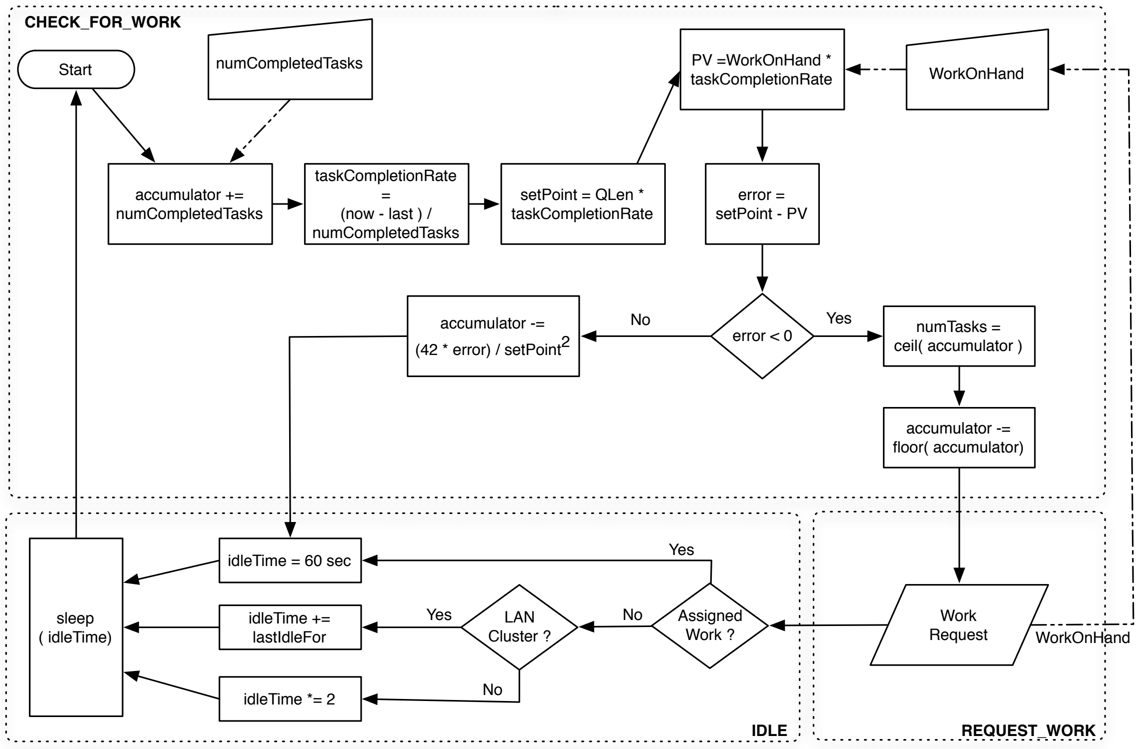

CHECK_FOR_WORK
Start
numCompletedTasks
PV =WorkOnHand * taskCompletionRate
WorkOnHand
accumulator += numCompletedTasks
taskCompletionRate = (now - last) / numCompletedTasks
setPoint = QLen * taskCompletionRate
error = setPoint - PV
accumulator -= (42 * error) / setPoint²
No
error < 0
Yes
numTasks = ceil(accumulator)
accumulator -= floor(accumulator)
idleTime = 60 sec
Yes
sleep (idleTime)
idleTime += lastIdleFor
Yes
LAN Cluster ?
No
Assigned Work ?
Work Request
WorkOnHand
idleTime *= 2
No
IDLE
REQUEST_WORK

Figure 4.3. Work Manager Controller Process Diagram

The *work manager* utilizes a feedback controller to regulate the task queue size, or work on hand. Figure 4.3 shows the controller process inlaid in the *work manager's* state machine. The controller's output variable is the task queue size, this is the variable we wish to affect. This variable is also referred to as the *process variable*. The controller's input variable, or the variable that it can alter to adjust the output, is the number of tasks to request. By regulating the number of tasks the *work manager* requests the queue size can be regulated. The initial queue size is zero, and the initial task completion rate is set to 60 seconds so that a node's initial work request is one task. As the processing speed of our node increases, more tasks will be requested and the queue size will increase. As the processing speed decreases, fewer tasks will be requested and the queue size will decrease.

Controllers compare the process variable with a *setpoint* to determine how to adjust the input variables. In the thermostat example, the setpoint is the desired room temperature and the process variable is the current room temperature. Here the setpoint is the desired queue size and the process variable is the current queue size. The setpoint is somewhat arbitrary in nature. We calculate our setpoint as the desired queue size, *QLen* in fig. 4.3, times the *task completion rate.* The *task completion rate* is the number of tasks completed per second, an observable value. Task completion rates less than or equal to zero are reset to the default value of 60, or 1 task per minute. Our process variable, PV in fig. 4.3, is the amount of work on hand, or current queue size, scaled in terms of the task completion rate. The upper bound for the process variable is set to 691200 seconds, or 8 days. Meaning at the current task completion rate, the maximum queue size will facilitate 8 days worth of work. The error term for the controller is the setpoint minus the process variable. The sign of error term is evaluated. A zero error term indicates the queue size is about right, and a negative value indicates the queue size is too large. In either case, a derivative term is applied to the accumulator and we proceed to the IDLE state. If the error term is positive, we have less than the desired amount of work on hand so we proceed to the REQUEST_WORK state. The accumulator indicates the amount of work that should be requested, after saving this value for use in the REQUEST_WORK state, the accumulator is reset.

In the REQUEST_WORK state, the *work manager* iterates through a list of available cluster managers making task requests over the network. It will continue iterating through the cluster manager list until it has reached the end of the list or until it has been assigned enough work to meet the expected demand as calculated in the CHECK_FOR_WORK state. The cluster manager list is a shared data object and is constructed by the *cluster browser* subsystem. If the end of the list is reached, a flag is set indicating that the *cluster browser* should begin browsing for clusters again. As tasks are assigned, entries are made in the datastore's *Task* table. Entries include information indicating the associated project, the assignment time, the due-time, the assigning cluster manager, and the task manifest. Each task's state is set to TASK_NEW. This prevents the task from being assigned until the required files are downloaded by the *download manager.*

In the IDLE state, the *work manager* sleeps for a variable period of time. If the *work manager* is entering the IDLE state from the CHECK_FOR_WORK state, the sleep duration is a constant 60 seconds. If the *work manager* is entering the IDLE state from the REQUEST_WORK state and work was assigned, the sleep duration is a constant 60 seconds. If the *work manager* is entering the IDLE state from the REQUEST_WORK state and work was not assigned, then the sleep duration is set according to a back-off algorithm. The back-off algorithm is used to give the cluster manager a chance to gather work before requesting work again. If the cluster manager resides on the same local area network as the client, as determined by its IP address, then a Fibonacci back-off is used. Otherwise, exponential back-off is used.

4.3.3 The Download Manager

The *Download Manager* is responsible for communicating with potential cluster managers and handling all file downloads. The download manager runs in its own thread. There are four states in the *download manager's* lifecycle (fig. 4.4): INIT, NEEDED_FILES, REQUEST_FILES, and IDLE.

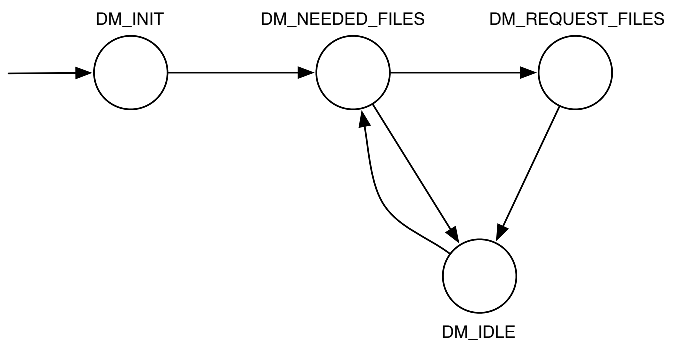


Figure 4.4. Download Manager State Machine Diagram

In the INIT state, the *download manager's* local variables are initialized, the download manager then enters the NEEDED_FILES state.

In the NEEDED_FILES state, the *download manager* queries the *Task* table for tasks in the TASK_NEW state. For each task in the TASK_NEW state, the task's manifest is parsed to determine the data files and executables needed by the task. For each required file, the *download manager* polls the *File Manager* framework to determine if the files have been registered. A registered file is a file that has been downloaded and associated with its project with an entry in either the *Executables* table or the *DataFiles* table. A list of files that are not registered is constructed and passed into the REQUEST_FILES state. If list is empty, then no files are needed and the *download manager* proceeds into the IDLE state instead.

In the REQUEST_FILES state, the *download manager* contacts the cluster manager which assigned the task and requests all files present in the file request list. Note that the request is not made from the cluster manager where the project originates. Data file requests are straight forward, but executables require extra effort. The *download manager* must request executables for all operating system and architecture platforms it wishes to complete the tasks on. For nodes acting as clients only, this is simply its own operating system and architecture platform. However, if this node is also acting as a cluster manager, it may need to request executables for its clients' platforms as well. Currently, the middleware takes the naive approach and requests all available versions of the executable for which it has at least one active client. For example, a Windows XP machine running on an x86 processor who has a single PowerPC Macintosh client and twelve XP clients would need to download two executables for each project's tasks: one executable for the x86/XP platform and one for the PowerPC/Macintosh platform. This would allow each task to be assigned to any of its associated clients, but given the ratio of platforms in this case, our approach could prove wasteful - depending on the performance of the single Macintosh client and the download or rotate frequency of the executables involved. The *download manager* could possibly benefit here by predicting which platforms it *should* request rather than requesting the executable for all client platforms. Further research will be needed in this area. The REQUEST_FILES state handles the file transfer using the same TCP connection as the file request. After each file has been downloaded, its MD5 checksum is calculated, if the transfer was successful the file is registered, otherwise the file is requested again at a latter point in time. After every needed file is downloaded for a task, the task's state is updated to TASK_READY which indicates that it is ready to be processed by a *Work Processing* thread or by a client node.

In the IDLE state, the download manager sleeps for one second and then re-enters the NEEDED_FILES state.

4.3.4 The Upload Manager

The *upload manager* is the module responsible for returning completed tasks and error messages to the cluster manager that assigned the work. The *upload manager* has four states (fig. 4.5): INIT, CHECK_RESULTS, RETURN_WORK, and IDLE.

In the INIT state, the *upload manager* initializes its local variables and then moves into the CHECK_RESULTS state.

In the CHECK_RESULTS state, the *upload manager* queries the datastore for tasks with the TASK_COMPLETED task state assigned. These are tasks that have been processed by the *work processing* threads and are packaged for return to the cluster manager which assigned them. A list of completed tasks and their result files is built from the query. This list is passed into the RETURN_WORK state.

In the RETURN_WORK state, the *upload manager* contacts each cluster manager in the list and returns all the result files, or error messages, for tasks assigned by that cluster manager. If a cluster manager is unavailable the results are stored for a future attempt. If a node is not able to return completed results within a week of their project designated due time, the results are discarded and no further attempt is made to return them. Likewise, the assigning cluster manager would reset the status of an unreturned result and reassign it. After all possible tasks or error messages are returned, the *upload manager* progresses into the IDLE state.

In the IDLE state the *upload manager* pauses for five minutes, then continues back into the CHECK_RESULTS state to check for more completed work.

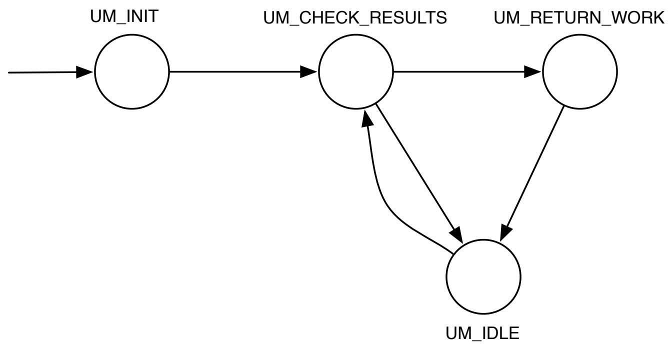


Figure 4.5. Upload Manager State Machine Diagram

<u>4.3.5 The Multicast Manager</u>

The *multicast manager* is a thread that listens for special multicast discovery messages. The packets contain a short *DISCOVERY* message and are sent out by nodes on a LAN to discover local cluster managers. When the *multicast manager* receives a *DISCOVERY* message, it responds with the amount of local and foreign work it has available to assign, the node's cluster manager port, a TCP port, and its IP address. Local work is work the machine is hosting, i.e. projects and tasks are created on that machine not another. Foreign work is work from any other machine, whether the cluster manager is located on the LAN or on the Internet.

4.3.6 The Cluster Browser

The *cluster browser* is responsible for locating and tracking available cluster managers. It ranks discovered cluster managers according to round-trip-time (RTT) heuristics. Local cluster managers, or cluster managers residing on the same network as the node, are preferred, followed by wide-area network cluster managers, or cluster managers hosted behind the same Internet router, and finally cluster managers from the Internet at large. The *cluster browser* has four states: INIT, LAN_BROWSE, INTERNET_BROWSE, and IDLE. Figure 4.6 shows the state machine diagram for the *cluster browser*.

In the INIT state, the *cluster browser* initializes its local variables before moving into the LAN_BROWSE state.

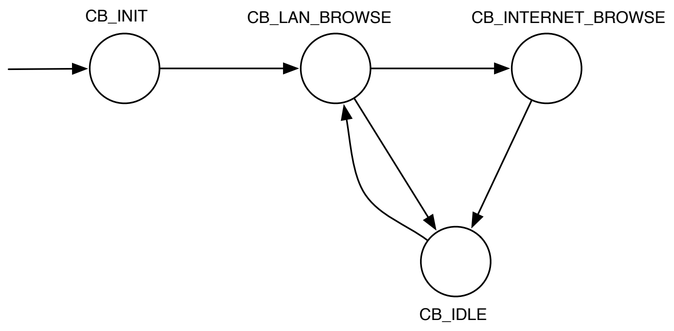


Figure 4.6. Cluster Browser State Machine Diagram

In the LAN_BROWSE state, the *cluster browser* searches the local area network (LAN) and the wide area network (WAN) for cluster managers with work. A simple multicast based discovery protocol is used during the LAN browsing phase. Discovery messages are broadcast on UDP port 52378 by default, though the port is configurable. The *cluster browser* sends out three DISCOVERY messages during LAN browsing, then waits for and logs responses.

The first discovery message is limited in scope to other devices on the same network hub or switch. This is indicated by setting the multicast scope to *link-local*. These machines represent the nearest possible neighbors on the physical network topology and typically have the lowest latency connections available on the local area network. The multicast scope of the second discovery message is set to *site-local.* Site-local multicast traffic can be forwarded to other network hubs and switches interconnected through local routers. Site-local messages likely reach all nodes residing on the same physical campus, because these nodes likely share a common router. These connections are low latency connections, though typically slower than link-local connections. Note, not all routers are configured to forward multicast traffic, and the extent to which they do varies widely from network to network. The multicast scope of the third discovery message is set to *organization-local.* These messages would potentially be forwarded across wide area network connections via WAN routers. These message can potentially traverse the breadth of the organization's network, but not beyond to the Internet. After sending the three discovery messages one second apart, the *cluster browser* waits for responses.

The DISCOVERY response messages contain several pieces of information, including: the cluster manager's IP address, the amount of locally hosted work available, the priorities of locally hosted work, and the amount of work originating elsewhere. This information is feed into a stochastic algorithm, which assigns a rank to each of the responding cluster managers. Cluster managers with locally hosted work are favored over those with work originating elsewhere. Likewise, cluster managers with more higher priority tasks are favored over those with lower priority tasks. The algorithm uses lottery scheduling to rank the cluster managers, appending them to the *cluster manager list* from lowest to highest ranking. This cluster manager list is shared with the *work manager*, which asks for tasks from cluster managers in order. When the *work manager* has exhausted the list of local cluster managers, it sets a flag to indicate an Internet browse is necessary. If the Internet browse flag is set, the *cluster browser* proceeds to the INTERNET_BROWSE state, otherwise it enters the IDLE state.

In the Internet browse state, the *cluster browser* searches for cluster managers on the Internet. The *cluster browser* may enter this state when there are either no local cluster managers available, or the local cluster managers have not provided work for the past hour. This last heuristic is used to prevent nodes from contacting Internet cluster managers when there are temporary work shortages among local cluster managers.

Multicast browsing would be the ideal algorithm for finding the nearest Internet cluster manager, because it would ensure the neighbor with fewest hops was discovered first. While protocols exist for forwarding multicast traffic over the Internet, they are frequently not supported by legacy Internet routers or not implemented by Internet Service Providers limiting their current usefulness. Various techniques for constructing Internet-scale one-to-many communication channels have developed to work around the current limitations. Discussion of these methods is considered beyond the scope of this work.

Our Internet clustering implementation will utilize the Internet Relay Chat protocol (IRC). IRC is readily available and facilitates a channel for one-to-many communication across the Internet. Essentially, the *cluster browser* will connect to an IRC channel, like *#simpleds*, where cluster managers with work to distribute will make advertisements. When the *cluster browser* connects to the channel, cluster managers seeking workers will write a message to the channel describing their IP address and the amount of work they have available. The contents of these advertisements will be nearly identical with their multicast counter parts. The *cluster browser* will parse the advertisements, and begin ranking the candidates according to their round-trip-times. The best candidates will be appended to the *cluster managers list* below the local cluster managers. When needed, the *work manager* will attempt to contact these cluster managers in the order they are ranked.

Once the Internet browsing algorithm has completed, the *cluster browser* toggles the both the LAN and Internet browsing flags off and proceeds to the IDLE state. Once in the IDLE state, the *cluster browser* remains idle until the LAN browse flag is again marked by the *work manager.* The flag is checked every ten seconds. Once the flag is marked, the *cluster browser* proceeds to the LAN_BROWSE state and the cycle continues.

4.3.7 The Client Manager

The *client manager* is the thread that listens for and responds to client requests. To function as a cluster manager in the SimpleDS architecture, the *client manager* thread must be running. The *client manager* listens on TCP port 52377 by default, but is configurable to any valid TCP port.

For stability purposes a message passing protocol was chosen for communication. All communication uses the same basic XML structure. The contents of messages are contained between <*message*> tags. Our convention is to use lowercase for all tag labels. The contents of each message is signed by the sending parties private key for the purposes of non-repudiation. The base-64 encoded signature is contained in a <*sig*> tag following the message. The entire message content is used to generate the signature, including the <*message*> tags. The current implementation, however, does not currently validate these signatures. In the future, a key-exchange protocol and signature validation should be implemented, to ensure the validity of executables being transfered.

Valid client requests include: MEMBERSHIP_REQUEST, WORK_REQUEST, RESULT_RETURN, and FILE_REQUEST. Valid response messages include: OK, DONE, FILE, EXEC, WORK, NO_WORK.

The MEMBERSHIP_REQUEST is the first contact between a cluster manager and a client node. In the current implementation, this message notifies the cluster manager that this client has associated with it. The only valid response is an OK message from the server. The design of this message leaves room for the cluster manager to deny permission to join the cluster. Figure 4.7 displays the format of a MEMBERSHIP_REQUEST message.

```
<message>
        <type>MEMBERSHIP_REQUEST</type>
        <from>username@org::fingerprint</from>
        <os>OS-VER-ARCH</os>
        <reputation> Values </reputation>
        <reputationSig> Sig </reputationSig>
</message>
<sig> MESSAGE SIGNATURE </sig>
```

Figure 4.7. Membership Request Message Format

The <*type*> tag indicates the message type. The <*from*> tag contains the node's identity string. The identity string is composed of three parts: the owner's chosen username, organization, and the node's key's hexadecimal fingerprint. The username and organization default to *anonymous* and *unknown* respectively, users are not required to change these. Each node generates a private and public key pair when it is installed, its public key fingerprint is used to identify the node. The <*os*> tag contains the node's operating system, operating system version, and CPU architecture. The <*reputation*> tag is a placeholder for the future implementation of a reputation system, described in (Gilbert, Abraham, Paprzycki, 2004).

In a WORK_REQUEST message, the client tells the cluster manager that it would like to be assigned work. The cluster manager may respond with either a list of assigned tasks or a NO_WORK message. If work is assigned, each assigned task is sent in an individual WORK message, to which the client may accept the assigned task or report an error using an OK message or an ERROR message. After the last task is transmitted, the server sends a DONE message to indicate the end of the list. The WORK_REQUEST format is shown in figure 4.8.

```
<message>
        <type>WORK_REQUEST</type>
        <from>anonymous@unknown:fingerprint</from>
        <os>OS-VER-ARCH</os>
        <reputation>REP</reputation>
        <reputationSig>REP_SIP</reputationSig>
        <numWorkUnits> number </numWorkUnits>
</message>
<sig> MESSAGE SIGNATURE </sig>
```

Figure 4.8. Work Request Message Format

The <*numWorkUnits*> tag contains an integer value, which is the number of tasks that the client would like to be assigned.

The RESULT_RETURN message indicates that a client wants to return a completed task, or a task that has erred out. The cluster manager may respond with either an OK message, in which case the client begins uploading the result file, or a LATER message containing a future window when the client should attempt to return results. The LATER message is not currently implemented, but it remains as a place-holder for result return scheduling.

The FILE_REQUEST message type indicates that a client needs a file from the cluster manager. The same message is used to request data files and executables. The <*os*> tag is used to indicate the executable platform the file is needed for executable files. If the client needs executables for multiple platforms, they must send an individual request for each desired platform. The cluster manager may respond to a FILE_REQUEST with a FILE, EXEC, or NO_SUPPORT response. The FILE response message indicates the cluster manager is returning a data file. The binary contents of the data file follow after the closing tag of the message signature, see figure 4.9.

```
<message>
        <type>FILE</type>
        <from> nodeid </from>
        <os> OS VERS </os>
        <project> project </project>
        <file> file name </file>
        <fileSize> size </fileSize>
        <fileSig>
                File Signature
        </fileSig>
</message>
<sig> MESSAGE SIGNATURE </sig>
BINARY FILE DATA ...
```

Figure 4.9. File Response Message Format

The EXEC response has the same form as the FILE message, but indicates that the server is returning an executable for the desired computing platform. The binary contents of the file follow the closing tag of the message signature. The NO_SUPPORT message indicates that the cluster manager does not have the requested file. If the requested file is required to complete an assigned task, the task will be erred out by the client node. The FILE and EXEC messages contain two notable features: the <*fileSize*> tag and the <*fileSig*> tag. The file size is the file size in octets and the file signature contains the MD5 hash of the file. For security reasons, this MD5 hash should be signed by the originating project's private key to ensure that the executable has not been tampered with, however, in the current implementation the signature is simply the MD5 sum of the file.

The OK message is an affirmative response to the immediately proceeding message. ERROR messages are negative responses to the immediately proceeding message. The TCP connection is closed immediately following an ERROR message.

The DONE massage is an informative message used to indicate the end of a list. Each list element are communicated individually, with the DONE message used to indicate the end of the list.

4.3.8 Support Frameworks

There are three supporting frameworks used by SimpleDS: the *task scheduler,* the *file manager*, and *the statistical engine.* Support frameworks do not have their own threads or lifecycles. Rather, they are objects abstracting interaction with particular database tables or wrapping complex tasks into simpler more manageable ones.

The *statistical engine* is used to track events of interest in the system. The only statistic of interest in the current implementation is the task completion rate. Every time a client returns a completed task, or a *work processor* thread completes a task, the function informTaskCompletedOK() gets called. The function makes note of the current time in seconds and makes an entry of this time into the statistics database. These time ticks are then used to determine the *task completion rate*, or the number of tasks completed per second, which is used by the *work manager* to determine whether or not to make a work request.

The *file manager* handles all files for a project. It tracks whether or not a given data file or executable is on-hand or needs to be downloaded by the *download manager*. Once a data file or executable is no longer needed, the file manager takes care of deleting the file and all file references. The file manager sets up and tears down the temporary work directories used by *work processors* to execute tasks. The *work* *processor* threads make calls to the *file manager* to setup temporary directories. The *file manager* makes sure that the work directories contain all the required files and returns the path to the *work processor.* Once the task has been executed, the *work processor* calls the *file manager* again to clean up the directory. All data input files and executables are deleted from the directory, the files left in the directory are then archived as a result, and a reference to the archived result file is saved for the *upload manager*. Basically, all filesystem calls are wrapped by the *file manager*. The *file manager* uses the Boost Filesystem library (BOOST, 2003) for all filesystem related system calls to ensure good portability between platforms.

The main purpose of the *task scheduler* is to assign tasks. It provides the function *getTasks*() to accomplish this goal. The *client manager* makes the request on behalf of clients sending WORK request messages, and the *work processors* call the function for themselves when they need another task to work on. The getTasks() function takes three arguments: a list of acceptable operating system platforms, the number of desired tasks, and the ID string of the requesting node. The task scheduler takes the operating system list and constructs a list of projects supporting those platforms. Work is assigned from this list of projects. If the available projects do not support any operating system on the input list, no work is returned. If the projects in the constructed list have differing project priorities, lottery scheduling is used to determine the priority level to assign from first. The probabilities .75, .18, .05, .01, and .01 are assigned to the priority levels 0, 1, 2 , 3, and 4 respectively. If there are not enough available tasks at the chosen priority level, the next level down is iteratively chosen until either enough work has been assigned or all priority levels have been visited. The *task scheduler* then returns a list of the assigned work. The status of each assigned task is changed from TASK_READY, to TASK_ASSIGNED to prevent the task from being assigned multiple times, though we plan on supporting task redundancy in the future, it is not currently implemented.

CHAPTER 5

EMPIRICAL EVALUATION

This chapter outlines our planned empirical evaluations. Each evaluation is presented in a single subsection. Each subsection will briefly describe the goal for the evaluation, provide a setup (where appropriate), a list of constants, a list of variables, a list of measurements, and a short procedural (where appropriate).

5.1 Lottery Scheduling Simulation for Task Assignment by Cluster Managers

Each cluster manager may host multiple projects with varying priorities. We will evaluate a stochastic lottery scheduling algorithm for priority scheduling at the cluster manager level. The goal of this simulation is to tune the parameters of our lottery scheduling algorithm for task assignment to find a balance between fairness and prioritization under ideal circumstances.

Constants. The simulation will have one cluster manager and 100 clients. Each client will have 100% availability. For simplicity, all clients will be assumed to have the same CPU rating. All tasks are created at time 0 and have the same duration - one simulated hour. There are three projects for each priority level. There are five priority levels, level one being the highest priority and five being the lowest. Each project has 1,000 tasks to complete, 15,000 tasks total.

Variables. The first variable is the ratio of lottery tickets from each priority level to the next. We will begin our investigation using the following three ratios: 2:1, 4:1, and *n*:(*n-1*). The ratio 2:1 is the ratio suggested by Waldspurger and Weihl. (2004). We choose one steeper ratio and one flatter ratio to begin our investigation. With the ratio 2:1 priority level 1 will be given 16 lottery tickets. Each subsequent priority level will be given half as many, level five having only one ticket. With the 4:1 scheme, priority level one will be given 256 tickets. Each subsequent level will be given 1/4 this number with priority level five having one ticket. With the *n*:(*n-1*) scheme, priority level 1 will have 5 tickets, level 2 will have 4 tickets, etc. Level 5 will be given 1 ticket.

The second variable is the selection algorithm for projects (tasks) with the same priority level. The simulation will cover *first-come-first-serve*, *round-robin*, and *random selection*. Random selection will simply choose one of the three projects with equal probability.

The third variable is the decision made in the event a given priority level has no tasks. There are two reasonable choices: choosing the next priority level up or choosing the next priority level down.

Measurements**.** The average turnaround time for tasks at each priority level will be calculated.

Procedure**.** Time begins at zero and increments linearly at one minute intervals. All clients arrive at time zero for task assignment. Each client is assigned one task at a time, returning for another task once the duration for the previous task is fulfilled. A random number is then generated (the lottery ticket). The range of the number is based on the ticketing scheme, the 2:1 scheme ranges between 0 and 30, the 4:1 scheme ranges between 0 and 339, and so on. The winner is chosen by the range of the value chosen. The simulation ends when all tasks are completed. A simulation run is performed for each combination of the three defined variables.

When all the tasks are the same duration and each priority level has the same number of tasks, all priority one tasks should finish before lower priority tasks. However, lower priority tasks should be given the opportunity to make progress. We will select the ratio scheme which best meets this constraint and implement this scheme into the middleware.

Results**.** There was no difference in performance associated with the out-of-task resolution algorithms in terms of average turnaround time per priority level. Therefore, for the discussion regarding average turnaround times, we will plot only the results for the scheduling scheme using the round-robin out-of-task resolution algorithm.

The scheduling schemes utilizing ratios 2:1 and 4:1 with *upward out-of-task resolution* exhibit priority inversions when the highest priority level has no tasks left to distribute (fig. 5.1). When the highest priority levels are out of work, the *upward out-of-task resolution* function looks for tasks on the lowest priority level. This behavior causes the lowest level priority projects to complete before some higher level priority projects. This is not the desired behavior when the task duration is constant and equal. We can rule out the use of these scheduling schemes.

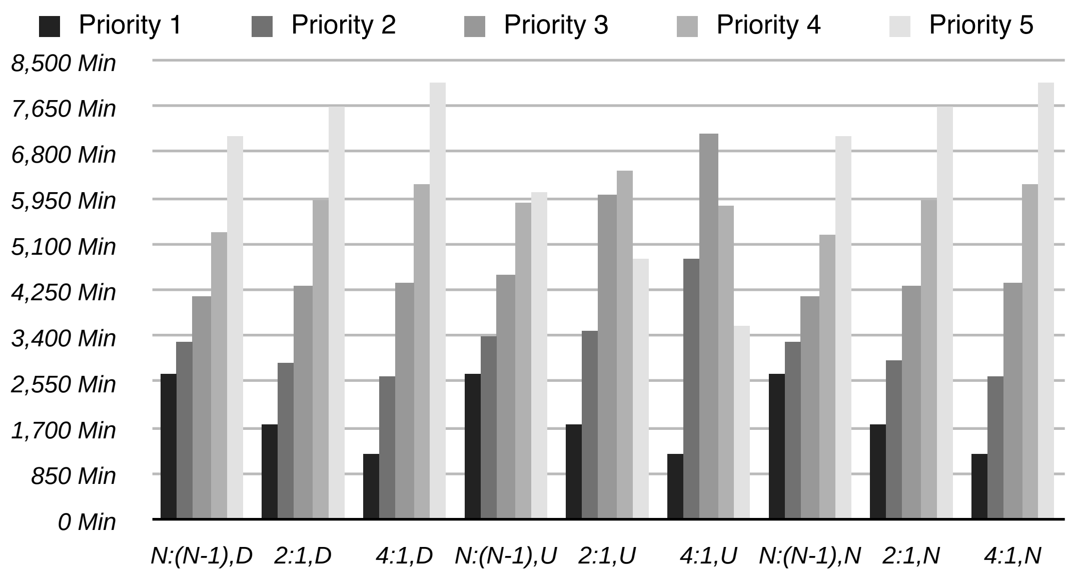


Figure 5.1. Average Turnaround Time Per Priority Level

All of the scheduling schemes using the ratio N:(N-1) lack sufficient distinction between priority levels. We would prefer a more significant distinction between priority levels than this scheme can provide, hence we will avoid using these schemes. We continue by examining the behavior of the remaining scheduling schemes against a first-in-first-out (FIFO) scheduling policy (Figure 5.2).

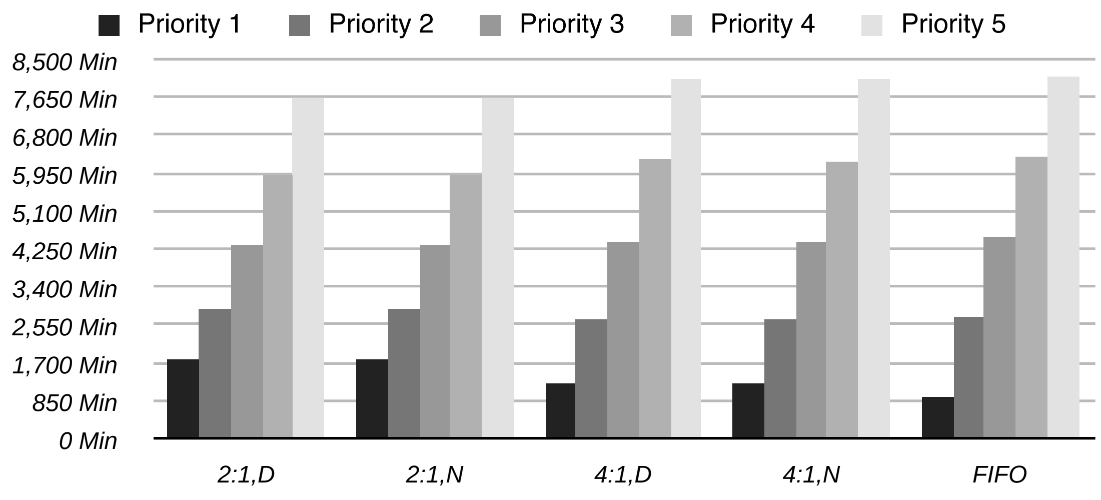


Figure 5.2. Comparing Lottery Scheduling Against FIFO Scheduling

The lottery scheduling scheme using a 4:1 ratio with either *nearest* or *downward* out-of-task-resolution is comparable to FIFO when all of the task durations are equivalent. The advantage to using a stochastic scheduling is apparent when the lower priority tasks have shorter durations. Because lottery scheduling gets these tasks assigned sooner, much like a shortest-job-first scheduler would do, projects with short tasks don't have to wait as long before completing. To demonstrate this, we will change the run time of all tasks from 60 minutes to 1 minute, except priority level 1 tasks, which will retain their 60 minute duration. Figure 5.3 shows the results for this scenario. We observe the lottery scheduling schemes as having a slight edge over FIFO in this case.

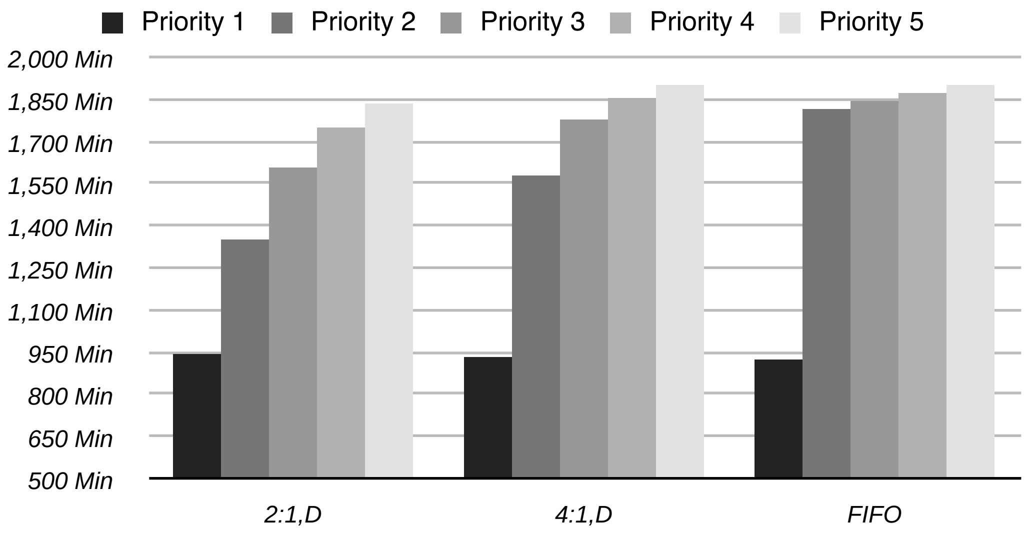


Figure 5.3. Comparing Lottery Scheduling Against FIFO With Shorter Task Times In Lower Priority Levels

Now we will discuss the affect the secondary scheduling algorithms have on the average TAT of tasks on the same priority level. The secondary scheduling algorithm is used to choose between multiple projects existing on the same priority level. Ideally, we would like projects on the same priority level to have equivalent average turnaround times. Figure 5.4 shows the average TAT of projects on priority level 1 for the primary lottery scheduling scheme using a 4:1 ratio, with *downward* out-of-task resolution.

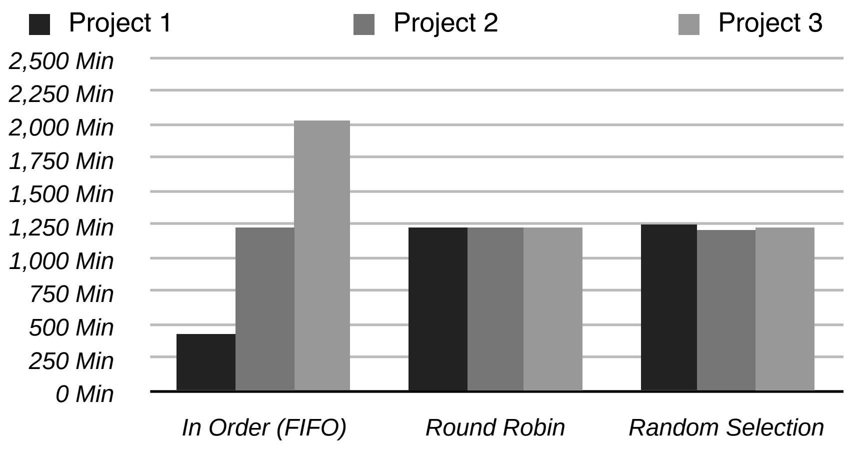


Figure 5.4. Performance of Secondary Scheduling Algorithms

We can see that choosing the *In Order* or *FIFO* scheduling algorithm for selecting a project among projects on the same priority level leads to undesirable behavior. Both the *Round Robin* and *Random Selection* algorithms display the desired behavior. Random selection has the advantage of not needing to reference the previously selected project in order to make the scheduling decision. We see this as an advantage in a highly concurrent scenario.

CHAPTER 6

CONCLUSIONS

We had high expectations at the onset of this project. As the project progressed and the number of bugs encountered became overwhelming, one thing became painfully clear: debugging multiprogrammed network applications is hard. These applications exhibit nondeterministic behavior. Some faults are simply not consistently reproducible, even in controlled environments. Further, any change to any aspect of the middleware required the system be fully retested to ensure correctness, making the debugging process even more intractable. Modularizing the system helped, but wasn't a panacea. Given the size of the project, we conclude that we grossly underestimated the amount of time needed to implement, test, and debug our middleware. Having said that, we still feel the work is an incremental step forward and was worth undertaking. Many valuable lessons were learned and will not be soon forgotten.

One valuable conclusion that we draw from this project is this: as hard as multiprogramming is, it can be leveraged to simplify a program's architecture. When used with appropriately modularized components, a multi-threaded application can actually be easier to implement and debug than its single threaded counter part, because it offers a vehicle for simplifying a system's state machines.

6.1 Future Work

There are many areas where this project can be improved upon. Our approach to nearly every system module provides opportunity for improvements and future work in many areas, including: file caching, data compression, and task scheduling. Here we outline some of what we feel could be the most interesting future work.

One of the larger problems with public-resource computing systems is ensuring the reliability of the results. The system must prevent inaccuracies introduced by calculation errors (e.g. from over-clocked CPUs) or cheaters. The current approach to ensuring accurate results is to have each task computed by three to five nodes and compare the results. Germain (2003) suggests that equivalent confidence levels can be achieved through statistical sampling of tasks, whereby a smaller portion of tasks would be rechecked by trusted sources and untrusted sources would have their work double-checked more frequently. This approach would tie in nicely to a reputation system or a blacklist solution. ZetaGrid uses some trust metrics to this end, but their exact algorithms are not available in the literature at this time. This is certainly an area worth further investigation.

In a hierarchical clustering approach to public-resource computing, as well as peer-to-peer approaches, each intermediate server has to anticipate future work-demand and attempt to have the work on hand to fill it. This is a deep and interesting produce-consumers problem providing ample opportunity for future work relating to the *work manager's* task scheduling.

Finally, our project's task scheduler is an area for further work. The current implementation is believed to be reasonable through comparison with a FIFO scheduler, however better approaches may exist and further work is suggested to this end. We feel the application of machine learning and other artificial intelligence approaches may be of particular interest here, since there is a good deal of inference taking place. Essentially, the job of deciding which tasks should be assigned to a particular client can be viewed as a classification problem. Likewise, comparing the speed or reliability of a node relative to the other participants is a classification problem. Our initial instinct here was to investigate the use of a distributed reputation system as an input to artificial intelligence approaches, our thoughts are outlined to some extent in (Gilbert, Abraham, Paprzycki, 2004).

REFERENCES

D. Abramson, R. Sosic, J. Giddy and B. Hall, "Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations", *The 4th IEEE Symposium on High Performance Distributed Computing*, Virginia, August 1995.

“Aglets,” IBM Corp., Internet: http://www.trl.ibm.com/aglets/. 2002.

ALICE: A Large Ion Collider Experiment at CERN LHC, Internet: http://aliceinfo.cern.ch. 2007.

V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira, "Characterizing Reference Locality in the WWW," *Proceedings of 1996 International Conference on Parallel and Distributed Information Systems (PDIS '96)*, pp. 92-103.

L. Amini and H. Schulzrinne, "Client Clustering for Traffic and Location Estimation," *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS’04)*, 730-737. March 2004.

D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, “SETI@home: An experiment in public-resource computing,” *Communications of the ACM*, Vol. 45, no. 11, 56-61, November 2002.

D. P. Anderson, "Public Computing: Reconnecting People to Science," *Proceedings of the Conference on Shared Knowledge and the Web*, Residencia de Estudiantes, Madrid, Spain, November 2003.

D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," *Proceedings of 5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, November 2004.

N. Andrade, W. Cirna, F. Brasileiro, P. Roisenberg, “OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing,” *Proceedings of the Ninth Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

The Anthrax Project, Internet: http://www.grid.org/projects/anthrax/. 2005.

The Apache HTTP Server Project, Internet: http://httpd.apache.org/. 2005.

Y. Aridor and D. B. Lange, “Agent Design Patterns: Elements of Agent Applications Design,” *Proceeding of the Second International Conference on Autonomous Agents (Agents '98)*, May 1998.

M. Arlitt, D. Krishnamurthy, J. Rolia, “Characterizing the Scalability of a Large Web-Based Shopping System,” ACM Transactions on Internet Techonolgy, Vol. 1, No. 1, Aug. 2001, pp. 44-69.

J. E. Baldeschwieler, R. D. Blumofe and E. A. Brewer. “ATLAS: An Infrastructure for Global Computing,” *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Cape Town, South Africa, Dec. 1999.

C. Baumer, M. Breugst, S. Choy, and T. Magedanz. “Grasshopper: a universal agent platform based on OMG MASIF and FIPA standards,” Technical report, IKV++ GmbH, 2000.

M. Beck, J.J. Dongarra, G.E. Fagg, G. Al Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S.L. Scott, and V. Sunderam, "HARNESS: A Next Generation Distributed Virtual Machine," International Journal on Future Generation Computer Systems, Elsevier Publ., Vol. 15, No. 5/6, 1999.

C.G. Bell, “Toward a history of (personal) workstations,” Proceedings of the ACM Conference on the History of Personal Workstations, January 1986.

Berkeley Open Infrastructure for Network Computing (BOINC), Internet: http://boinc.berkeley.edu/. 2005.

T. Berners-Lee, R. Fielding, L. Masinter, “Uniform Resource Identifiers (URI): General Syntax”, IEEE Network Working Group, RFC 2396, August 1998.

J. Best, W. Chamberlain, S. Maene, “Astrophysical Computational Research in a Small College Setting,” Journal of Comptuing Sciences in Colleges, Vol. 17, No. 3, Feb 2002, pp. 194-202.

S.N. Bhatt, F.R.K. Chung, F.T. Leighton, and A.L. Rosenberg, “On Optimal Strategies for Cycle-Stealing in Networks of Workstations,” IEEE Transactions Comp., Vol. 46, No. 5, May 1997, pp. 545-557.

Boost C++ Libraries, Internet: http://www.boost.org/. 2003.

D.A. Butterfield and G.J. Popek, "Network tasking in the Locus distributed UNIX system," Proceedings of the Summer USENIX Conference, June, 1984, pp. 62-71.

W. Bux. “Local-area subnetworks: A performance comparison,” IEEE Transactions. Communications COM. Vol. 29, No. 10 (Oct.), 1981.

R. Buyya, D. Abramson, and J. Giddy, "A Case for Economy Grid Architecture for Service Oriented Grid Computing", *10th IEEE International Heterogeneous Computing Workshop (HCW 2001), In conjunction with IPDPS 2001*, San Francisco, California, U.S.A, April 2001.

R. Buyya, D. Abramson, and J. Giddy, “Nimrod-G Resource Broker for Service-Oriented Grid Computing”, IEEE Distributed Systems Online, in Volume 2 Number 7, November 2001.

R. Buyya, S. Vazhkudal, “Compute Power Market: Towards a Market-Oriented Grid," Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '01), May, 2001.

The Cancer Research Project, Internet: http://www.grid.org/projects/cancer/. 2005.

G. Candea and A. Fox, “Crash-Only Software,” *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

M. Castro, P. Druschel, Y.C. Hu, A. Rowstron, "Exploiting network proximity in peer-to-peer overlay networks," *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo 2002)*, 2002.

M. Castro, P. Druschel, Y.C. Hu, and A. Rowstron, "Topology-aware routing in Structured peer-to-peer overlay networks," *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo 2003)*, 2003.

J.A. Chambers, R.V. Poore, “Comuter Networks in Higher Education: Socio-economic-political Factors,” Communications of the ACM. Vol. 18, No. 4, April 1975, pp. 193-199.

S. Chapin, J. Karpovich, A. Grimshaw, “The Legion Resource Management System”, *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing*, April 1999.

D. Chavey, “A Multi-purpose Computer Lab for a Small College,” Proceedings of the Twenty-third Annual ACM Conference on User Services: Winning the Networking Game, St. Louis, Missouri, USA, 1995, pp. 67-72.

S. Cheshire, “Multicast DNS,” Draft RFC, June, 2005.

B.O. Christiansen, P. Cappello, M.F. Ionescu, M.O.Neary, K.E. Schauser, and D. Wu, "Javelin: Internet-based parallel computing using java," ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation, June 1997.

W. Cirne, and K. Marzullo, “Open Grid: A user-centric approach for grid computing,” *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*, 2001.

ClimatePrediction.net, Internet: http://climateprediction.net/. 2005.

B. Cohen, “Incentives Build Robustness in BitTorrent,” Workshop on Economics of P2P Systems. June 2003.

“Common Vulnerabilities and Exposures: the Standard for Information Security Vulnerability Names,” Internet: http://www.cve.mitre.org/. 2005.

M. Costa, M. Castro, A. Rowstron, P. Key, "PIC: Practical Internet coordinates for distance estimation", *Proceedings for the Twenty-fourth International Conference on Distributed Systems*. Tokyo, Japan, March, 2004.

D. Côté, "Simple: XGrid agent for UNIX architectures," Internet: http://www.novajo.ca/simple/archives/000026.html. June, 2004.

R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, "Practical, Distributed Network Coordinates," ACM SIGCOMM Computer Communication Review, Vol. 34, No. 1, January, 2004, pp113-118.

S. Deering, “Host Extensions for IP Multicasting,” RFC 1112, Aug. 1989.

Distributed.net, Internet: http://www.distributed.net/. 2005.

Earth Simulator Group, Internet: http://www.es.jamstec.go.jp. 2005.

K. Egevang, and P. Francis, “The IP Network Address Translator (NAT),” RFC 1631, May 1994.

Einstein@home, Internet: http://www.physics2005.org/events/einsteinathome/index.html. 2005.

The Ethernet: A Local Area Network Data Link Layer and Physical Layer Specifications, Sept. 30. 1980. DEC, Intil Corp. Xerox Corp. Maynard, Mass.

G. Fagg, A. Bukovsky, J. Dongarra, "HARNESS and Fault Tolerant MPI," Parallel Computing, Vol. 27, No. 11, October 2001, pp. 1479-1496.

FightAids@home, Internet: http://fightaidsathome.scripps.edu/. 2005.

Find-a-Drug, Internet: http://www.find-a-drug.org.uk. 2005.

Folding@home, Internet: http://folding.stanford.edu. 2005.

M. Forum, "MPI: A Message-Passing Interface Standard," Technical report, University of Tennessee, June 1995.

I. Foster and C. Kesselman, “Globus: A metacomputing infrastructure toolkit,” *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, 115-128, 1997.

I. Foster and C. Kesselman, Eds., The Grid: Blueprint for a Future Computing Infrastructure. Morgan Kaufmann Publishers, 1999.

A. Fox, S.D. Gribble, Y. Chawatha, E.A. Brewer, P. Gauthier, "Cluster-based Scalable Network Services," *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, Saint Malo, France, 1997, pp. 78-91.

P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, L. Zhang, "IDMaps: A Global Internet Host Distance Estimation Service," IEEE/ACM Transactions on Networking, Vol. 9, No. 5, October, 2001.

M. Freedman and D. Mazieres. "Sloppy hashing and self-organizing clusters". In Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, February 2003.

M.J. Freeley, B.N. Bershad, J.S. Chase, and H.M. Levy, "Dynamic Node Reconfiguration in a Parallel-Distributed Environment," Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM Press, New York, NY, USA, 1991, pp 114-121.

J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," Cluster Computing, Vol. 5, No. 3, Springer Science, July 2002. pp 237-246.

V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519 "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy," September 1993.

S. Gammill, "Tomorrow the World: Establishing an Internet Connection at a Small College," *Proceedings of the Eighteenth Annual Conference on User Services (ACM SIGUCCS)*, Cincinnati, Ohio, USA, 1990, pp. 137-140.

Genome@home, Internet: http://www.stanford.edu/group/pandegroup/genome/. 2005.

C. Germain, "Result Checking in Global Computing Systems," *Proceedings of the 17th Annual International Conference on Supercomputing*.San Francisco, California. 2003.

A. Gilbert, J. Thomas, and I. Jonyer, "Modeling Work Flow in Hierarchically Clustered Distributed Systems," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04)*. Las Vegas, Nevada. June 2004.

A. Gilbert, A. Abraham, M. Paprzycki, "A System for Ensuring Data Integrity in Grid Environments," *Proceedings of the International Conference Information Technology (ITCC'04)*, Las Vegas, Vol. 1, 435-439, 2004.

The gLite Middleware Project, Internet: http://glite.web.cern.ch/glite/. 2005.

The GNU Compiler Collection, Internet: http://gcc.gnu.org/. 2007.

M. Graube. "Local area nets: a pair of standards", IEEE Spectrum (June), 60-64. 1982.

P. Gray, and V.S. Sunderam, "Metacomputing with the IceT System," International Journal of High Performance Computing Applications, Vol. 13, No. 3, 1999, pp. 241-252.

Great Internet Mersenne Prime Search, (GIMPS), Internet: http://www.mersenne.org/prime.htm. 2005.

Grid MP Middleware, Internet http://www.ud.com/solutions/deploy/mp_enterprise.htm. 2005.

R. Groves, "CppSQLite - C++ Wrapper for SQLite," Internet: http://www.codeproject.com/database./CppSQLite.asp. 2004.

Z. Haas, D.R. Cheriton, "A Case For Packet Switching in High-Performance Wide-Area Networks," Proceedings of the ACM Workshop of Frontiers in Computer Communications Technology, ACM SIGCOMM Computer Communication Review, Vol. 17, No. 5, Aug. 1987.

B. Huffaker, M. Fomenkov, D. Plummer, D. Moore, and K. Claffy, "Distance Metrics in the Internet," *Proceedings of the International Telecommunications Symposium (ITS'02)*, 2002.

The Human Genome Project, Internet: http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml. 2005.

The Human Proteome Folding Project, Internet: http://www.grid.org/projects/hpf/about.htm. 2005.

O.C. Ibe, H. Choi, and K.S. Trivedi, "Performance evaluation of client-server systems," IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 11, Nov. 1993, pp. 1217-1229.

"IEEE Project 802", Local Network Standards. Institute of Electrical and Electronic Engineers, New York. 1983.

IEEE Std 1003.1-2001 Standard for Information Technology - Portable Operating System Interface (POSIX) Base Definitions, Issue 6. IEEE, New York, NY, USA, 2001.

M. Izatt, P. Chan and T. Brecht, "Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications", *Proceedings ACM 1999 Java Grande Conference*, pp. 15-25, June 1999.

G. Judd, M. Clement, Q. Snell, "DOGMA: Distributed Object Group Management Architecture" Concurrency: Practice and Experience, Vol. 10, No. 11-13, Sept. 1998, pp. 977-983.

E. Jul, H. Levy, N. Hutchinson, and A. Black, "A Fine-grained mobility in the Emerald System," ACM Transactions on Computer Systems, 6(1):109-133, February 1988.

N.T. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and S. Lacour, "A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids," Technical report ANL/MCS-P948-0402, Mathematics and Computer Science Division, Argonne National Laboratory, USA, April 2002.

K. Krauter, R. Buyya, and M. Maheswaran, "A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing", International Journal of Software: Practice and Experience (SPE), ISSN: 0038-0644, Volume 32, No. 2, pp 135-164, Wiley Press, USA, February 2002.

B. Krishnamurthy and J. Wang, "On Network-Aware Clustering of Web Clients," in Proc. ACM SIGCOMM, August/September 2000.

The Large Hadron Collider (LHC) Project, Internet: http://lhc.web.cern.ch. 2007.

The Lattice Project, Internet: http://lattice.umiacs.umd.edu/. 2005.

LHC@home, Internet: http://athome.web.cern.ch/athome/. 2005.

M. J. Litzkow, "Remote Unix: Turning Idle Workstations into Cycle Servers," Proceedings of the Summer 1987 Usenix Conference, June, 1987, pp. 381-384.

M.J. Litzkow, "Condor - a hunter of idle workstations," Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, CA, USA, June 1988. pp. 104-111.

L. Loewe, "evolution@home: Experience with work units that span more than 7 orders of magnitude in computational complexity," *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, 2002.

M.J. Lorence, and M. Satyanarayanan, "IPwatch: a tool for monitoring network locality," ACM SIGOPS Operating Systems Review, Vol. 24, No. 1, pp 58-80, ACM Press, New York, NY, USA, January, 1990.

L. Massoulie, A.M. Kermarrec, and A.J. Ganesh, "Network awareness and failure resilience in self-organising overlay networks," Proceedings of the 22nd International Symposium on Reliable Distributed Systems, October, 2003. pp 47-55.

Message Passing Interface (MPI), Internet: http://www-unix.mcs.anl.gov/mpi. 2005.

R.M. Metcalfe, and D.R. Boggs. "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*. Vol. 19, No. 7 (July). 395-404. 1976.

Microsoft Visual Studio, Internet: http://msdn2.microsoft.com/en-us/vstudio/default.aspx. 2007.

The MITRE Corporation, Internet: http://www.mitre.org/. 2005.

M.O. Neary, B.O Christiansen, P. Cappello, and K.E. Schauser. "Javelin++: Scalability Issues in Global Computing," Proceedings of the ACM Java Grande 1999 Conference, San Francisco, California, June 12-14, 1999.

M O. Neary, A Phipps, S Richman, and Peter Cappello, "Javelin 2.0: Java-based parallel computing on the Internet" Proceedings Euro-Par 2000 Parallel Processing, pp. 1231-1238. 2000.

MySQL Open Source SQL Database, Internet: http://www.mysql.com/. 2005.

J.E. Nelno, C.M. Woodside, D. Petriu, and S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendezvous Networks," IEEE Transactions on Software Engineering, Vol. 21, No. 9, Sept. 1995, pp. 776-782.

T.S.E. Ng and H. Zhang, "Predicting Internet Network Distance with Coordinate-based Approaches," Proceedings of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, Vol. 1, June 2002, pp. 170-179.

D.A. Nichols, "Using Idle Workstations in a Shared Computing Environment," Proceedings of the 11th ACM Symposium on Operating Systems Principles, Nov., 1987.

M.A. Olson, K. Bostic, M. Seltzer, "Berkeley DB," Proceedings of the Paper - 1999 USENIX Annual Technical Conference, June 1999, pp. 183-192.

Oxford University's Centre for Computational Drug Discovery, Internet http://www.chem.ox.ac.uk/curecancer.html. 2005.

P. Padala, C. Harrison, N. Pelfort, E. Jansen, M. Frank, C. Chokkareddy, "OCEAN: The Open Computation Exchange and Arbitration Network, a Market Approach to Meta Computing," *Proceedings of the Second International Symposium on Parallel and Distributed Computing*, Oct. 2003.

V. Pande, et al. "Atomistic Protein Folding Simulations on the Submillisecond Time Scale Using World-wide Distributed Computing," Biopolymers, Vol. 68, 91-109, 2003.

Parallel Virtual Machine (PVM), Internet: http://www.csm.ornl.gov/pvm/pvm_home.html. 2005.

C. Parnot, "Xgrid@Stanford," Internet: http://cmgm.stanford.edu/~cparnot/xgrid-stanford/index.html. 2005.

A. Patrizio, "Genome Effort Hits Home," *Wired*, February, 2001. Available: http://wired.com/news/technology/0,1282,41842,00.html.

M. Pias, J. Crowcroft, S. Wilbur, T. Harris, S. Bhatti, "Lighthouses for scalable distributed location," Proceedings of the Second International Workshop on Peer-to-Peer Systems, Berkeley, CA, USA, February, 2003.

Predictor@home, Internet: http://predictor.scripps.edu/. 2005.

Python Programming Language, Internet: http://www.python.org/. 2005.

S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," Proceedings of ACM SIGCOMM, Aug. 2001.

S. Ratnasmay, M. Handley, R. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection." Proceedings of the Twenty-first IEEE INFOCOM, New York, N.Y., June 2002.

Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear, "Address Allocation for Private Internets," RFC 1918, February 1996.

M. Rose, "The Blocks Extensible Exchange Protocol Core," RFC 3080, March 2001.

A.L. Rosenberg, "Optimal Schedules for Cycle-Stealing in a Network of Workstations with a Bag-of-Tasks Workload," IEEE Transactions on Parallel and Distributed Systems, Vol. 13, No. 2, IEEE Press, Piscataway, NJ, USA, Feb. 2002, pp 179-191.

R. Rosenthal. The selection of local area computer networks. NBS Special Publication. 500-96. National Bureau of Standards, Washington, D.C., November, 1982.

M.A. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," IEEE Network, Vol. 15, No. 2, March/April 2001, pp. 8-23.

G.W. Scragg, "A Crisis in Computer Science Education at Liberal Arts Colleges," ACM SIGSCSE Bulletin, Vol. 19, No. 2, June 1987, pp. 36-42.

SETI@home 3.03 release notes. 2004.

SETI@home, Internet: http://setiathome.ssl.berkeley.edu/. 2005.

SETI@home forums, Internet: http://setiathome.berkeley.edu/forum_thread.php?id=23378. 2005.

J.F. Shoch. An annotated bibliography on Local Computer Networks. Xerox Palo Alto Research Center, Palo Alto, California, April, 1980.

J. Shoch and J. A. Hupp, "The 'Worm' Programs - Early Experience with a Distributed Computation." Communications of the ACM, March 1982. Vol. 25, No. 3, 172-180.

SimpleDS Middleware Project, Internet: http://www.breakingrobots.net/projects/SimpleDS/. 2005

The Smallpox Project, Internet: http://www.grid.org/projects/smallpox/. 2005

N. Stankovic and K. Zhang, "A Distributed Parallel Programming Framework," IEEE Transactions on Software Engineering, Vol. 28, No. 5, 2002, pp. 478-493.

T. Sterling, D. Savarese, D. Becker, D. J. Becker, J. E. Dorband, U. A. Ranawake, C. V. Packer, "Beowulf: A Parallel Workstation For Scientific Computation," *Proceedings of the 24th International Conference on Parallel Processing (ICPP)*, Vol. 1, 11-14, August 1995.

W. Sullivan III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson, "A new major SETI project based on project serendip data and 100,000 personal computers," Astronomical and Biochemical Origins and the Search for Life in the Universe, *Proceedings of the Fifth International Conference on Bioastronomy*, Editrice Compositori Publ., Bologna, Italy, 1997.

V. A. Sunderam, "PVM: A Framework for Parallel Distributed Computing." Concurrancy - Practice and Experience. Vol. 2, No. 4. 1990. 315-339.

SQLite Embeddable SQL database, Internet: http://www.sqlite.org. 2005.

H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "Network topologies, power laws and hierarchy," Tech. Rep. TR01-746, Technical Report, University of Southern California, 2001

T. Tannenbaum, and M. Litzkow, "The Condor distributed processing system," Dr. Dobb's Journal, February 1995.

M. M. Theimer, K.A. Lantz, and D.R. Cheriton, "Preemptable remote execution facilities for the V-system," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, Orcas Island, Washington, USA, Dec., 1985. pp. 2-12.

M. M. Theimer, "Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems," Ph.D. Th., Stanford University, June 1986. Available as Stanford Computer Science tech. report STAN-CS-86-1128.

M.M. Theimer, and K.A. Lantz, "Finding idle machines in a workstation-based distributed system," *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp 112-122. IEEE Computer Society, June 1988.

Top Five-Hundred Super Computer List, http://www.top500.org/lists/2004/11/. 2005.

United Devices, Internet http://www.ud.com/home.htm, 2005.

W. Vogels, R. van Renesse, K. Birman, “The Power of Epidemics: Robust Communication for Large-Scale Distributed Systems,” ACM SIGCOMM Computer Review, Vol. 33, No. 1, ACM Press, NY, January 2003. pp. 131-135.

M. Wahl, T. Howes, S. Kille, “Lightweight Directory Access Protocol (v3),” RFC 2251, December 1997.

C.A. Waldspurger, W.E. Weihl, “Lottery Scheduling: Flexible Proportional-Share Resource Management,” *Proceedings of the First Symposium on Operating Systems Design and Implementation*, USENIX Asssociation, Monterey, California, USA, Nov. 1994.

R. Winter, T. Zahn, J. Schiller, "Topology-Aware Overlay Construction in Dynamic Networks," *Proceedings of the Third International Conference on Networking (ICN 2004)*, Gosier, Gaudeloupe, French Caribbean, February/March 2004.

XGrid, Internet: http://www.apple.com/macosx/features/xgrid/. 2005.

X3.4 1986. American Standard Code for Information Interchange (ASCII) as defined by the American National Standards Institution standard X3.4-1986.

T. Zahn, R. Winter, J. Schiller, “Simple, Efficient Peer-to-Peer Overlay Clustering in Mobile, Ad-Hoc Networks,” *IEEE International Conference on Networks (ICON 2004)*, Singapore, November 2004.
“ZetaGrid: The Grid for Everybody,” Internet: http://www.zetagrid.net/

X.Y. Zhang, Q. Zhang, Z. Zhang, G. Song, and W. Zhu, "A construction of locality-aware overlay networks: mOverlay and Its Performance," IEEE Journal on Selected Areas in Communications, Vol. 22, No. 1, January, 2004.

APPENDIX A

SimpleDS USER MANUAL

Here we describe the system interface from a user's perspective. For demonstrative purposes, we will walk through creating an example project and preparing a few tasks for distribution. In this case, we will be creating a project to process 100 data files with a single executable. This kind of application belongs to the Single Program Multiple Data (SPMD) class of applications.

A.1 Creating Projects

A project is created from the command line using the *sdsaddproj* tool. Projects are referenced by unique a Universal Resource Identifier (URI) (Berners-Lee, Fielding, Masinter, 1998). Currently, the URI does not need to reference an actual web page, however, in the future a web page describing the project may be required. The following is the syntax for the *sdsaddproj* tool.

```
sdsaddproj <URI> <description> <email contact> [<priority>] [<redundancy count>]
```

The first argument is the project's reference URI. The second argument is a short description of the project. The third argument is a contact email address for the project. The fourth argument is optional and indicates the project priority, this value defaults to one. The final argument is also optional, it is the task redundancy value. This value also defaults to one and is ignored in the current implementation.

For our example, we will use the URI *http://www.agentlab.net/projects/example1*, a short description, and my email address, leaving the project priority and redundancy values at the default settings.

```
./sdsaddproj http://www.agentlab.net/projects/example1 “This is an example project”
austirg@cs.okstate.edu
```

Executing the command creates the directory $SDS_HOME/Project/<project name> for the project's files, where $SDS_HOME is the install path for SimpleDS, and <project name> is the project's short name. Short names are used because URIs do not represent valid path names on many operation systems. A project short name is derived from the project's URI. The short name is the value after the last slash in the URI. *E.g.,* the URI http://www.agentlab.net/projects/example1 would yield a short name *example1*. The use of project short names introduce the possibility of name collisions among projects. For example, the projects http://www.agentlab.net/projects/example1 and http://www.okstate.edu/SimpleDS/Projects/Whereever/example1 would cause a short name collision. A method for resolving short name collisions is planned but not currently implemented. After creating a project reference, data files and executables may be added to the project.

A.2 Registering Executables

Executables are expected to be statically compiled, no library support is provided by the middleware. Executables are registered using the *sdsaddexec* tool.

```
sdsaddexec [--no-cache] <URI> <name> <path>
sdsaddexec [--no-cache] --os=<OSTriplet> <URI> <name> <path>
```

The **<URI>** argument contains the project's URI. The **<name>** argument is the executables reference name within the project. The reference name need not be the same as the executable's actual name, however all future references to the executable must be made by the name supplied here. The **<path>** argument is a path to the executable being registered. During the registration process, a compressed copy of this file is made into $SDS_HOME/Projects/<project name>/Bin/<os name>/<os arch>/, where $SDS_HOME is the installation path of SimpleDS, <project name> is the project's short name, <os name> is the operating system name the executable is compiled for, and <os arch> is the CPU architecture the executable is compiled for.

The **--no-cache** flag tells the tool to inform clients not to cache the executable. Executables are cached by default according to the algorithm described in *section 3.5*. For our example application we want the executables cached. For an MISD or MIMD application, executables would be used only once, hence we would employ the **--no-cache** flag for that class of applications.

By default, *sdsaddexec* assumes that the executable being registered is compiled for the operating system hosting the project. E.g. if the project were running on Linux, then the tool assumes that the executable is a Linux executable. This behavior is changed by specifying the **--os** flag with an operating system triplet as an argument. An operating system triplet is a string in the form **<on name>-<os ver>-<os arch>**. The **<os name>** portion of the triplet describes the name of the operating system the file is compiled for, *e.g.* Linux, Darwin, Windows, etc. The **<os arch>** describe the CPU architecture the executable is compiled for. E.g., powerpc, x86, sparc, etc. The **<os ver>** portion is a string describing the version of the operating system the system is compiled for. *E.g.*, 8.2.0, Win32, 2.4.24-1, etc. This string is operating system dependent, and the system will work out operating system compatibility issues automatically. For example, an executable compiled for Linux 2.4.16 could also be run on 2.4.18 without issues but might not run on Linux 2.2.0. For the most part, upward compatibility will be assumed. That is, if an executable is registered for Linux 2.4.14, then it will be assumed compatible for 2.4.15, 2.4.16, 2.6.18 etc. Any exceptions will be handled on a case by case bases. Note that the current implementation ignores the OS version string and assumes complete compatibility between operating system versions.

To demonstrate, we will register two executables, one executable will be a Windows executable and the other will be for Linux. Both executables will perform the same operation, so they will be given the reference name *exec1*. For demonstrative purposes, assume the Linux executable resides in the directory /usr/local/share/myLinuxExec, and that the Windows executable resides in the directory /usr/local/share/myWindowsExec. Here, we show the executables being registered when the hosting system is Linux.

```
./sdsaddexec http://www.agentlab.net/projects/example1 exec1 /usr/local/share/myLinuxExec

./sdsaddexec --os=Windows-WIN32-x86 http://www.agentlab.net/projects/example1 exec1
        /usr/local/share/myWindowsExec
```

A.3 Registering Data Files

Data files are registered to a project using the *sdsadddata* tool. The syntax of this tool follows:

```
sdsadddata [--cache] [--update] [--bundle] <URI> <name> {<name>}
```

Again, **<URI>** specifies the project's URI. The **<name>** argument is the data file's reference name and its actual path on the system. If the file path contains directories, they will be removed and only the filename will be used. Tasks in the system will make reference to this file by this value. A compressed copy of the data file is made in $SDS_HOME/Project/<project name>/Data/, where $SDS_HOME is the installation path for SimpleDS and <project name> is the short name of the project URI. Data files are not cached by default. To cache a data file, for example a configuration file that will be used by multiple tasks, the **--cache** flag is specified. Otherwise, a data file is used by a task, then delete from the filesystem. The **--update** flag is used to tell the system to replace the current file (of the same name) with a newer version.

For our running example of creating a project with 100 data files to be processed, we would need to register each of the files using *sdsadddata*. A few examples are shown below:

```
./sdsadddata http://www.agentlab.net/projects/example1 data1
./sdsadddata http://www.agentlab.net/projects/example1 data2 data3 data4
./sdsadddata --cache http://www.agentlab.net/projects/example1 config1
```

Here we have registered the first four data files and the configuration file *config1* for use in the system. We can see this tool works well for one or two files, but registering 100 files this way would be tedious. Many operating system shells provide *wildcard expansion* features. The *sdsadddata* tool will capitalize on this feature where possible. In this example, all the files matching the pattern *data** would be imported to the project.

```
./sdsadddata http://www.agentlab.net/projects/example1 data*
```

Alternatively, an entire directory of data files can be imported into the system by passing a directory path as the name argument instead of a file name. Here, we are importing all the contents of the directory */home/austirg/mydata*:

```
./sdsadddata http://www.agentlab.net/projects/example1 /home/austirg/mydata/
```

This tool will also support bundling multiple data files into one compressed archive. In this case the **--bundle** option would be specified along with a directory path. Then all of the files contained in the directory would be compressed into one archive named after the directory.

```
./sdsadddata --bundle http://www.agentlab.net/projects/example1 /home/austirg/bundle1/
```

Finally, multiple bundles could be created simultaneously using wildcard expansion.

```
./sdsadddata --bundle http://www.agentlab.net/projects/example1 /home/austirg/bundle/*
```

In this case, any directory name matched by the wildcard * would become bundles, and any file names matched by the wildcard expansion would be imported as single files.

A.4 Creating Tasks

Tasks are created using the *sdsaddtask* tool. There are two required arguments, the URL of the project the task is being created under and the quoted manifest string describing the task:

```
./sdsaddtask http://www.agentlab.net/projects/example1 "MANIFEST STRING"
```

The command returns 0 if the task is created successfully, and returns -1 if there is an error. Possible errors may include an invalid task manifest string, an unregistered project, or a manifest string containing references to files that haven't been registered. The format of the manifest string is described in the following section.

A.5 Example Task Manifests

Task manifests are strings that describe tasks. They describe the components needed by a task and how these components relate to each other. Task components include data files, executables, operating systems/architecture constraints, compilers and/or interpreters needed, and output files created as a result of execution. The task manifest format is relatively straightforward but flexible. Rather than present a formal definition of the manifest grammar, we demonstrate the grammar with examples. At times, our examples will refer back commands from section A.3.

A.5.1 Simplest Example

The simplest possible manifest string looks like this:

```
data:( data1 )
exec:( exec1 ):( Darwin-8.0.0-powerpc )
{exec1}:( data1 ):()
```

Note we are adding return characters for readability, but these are not required by the system. The data declaration **data:( )** declares what data files the task will use. The example above requires one data file: *data1.* The data file name here refers to data files we registered previously using the *sdsadddata* tool, refer to section A.3 for a review.

The next section is the executable declaration. This section defines the executables needed by the task and the architectures the task can be executed on. This task requires an executable referred to as *exec1*.

The only supported platform is Apple's Macintosh operating system version 10, or Darwin 8 as reported by the unix command *uname -s* and *uname -r.*

The final section in this example declares the relationship between the data files and the executables. This example is equivalent to the following DOS or UNIX command lines: "exec1 data1". It simply says run *exec1* with the file *data1* as the only argument. All task output will be redirected to a file called *OUTPUT*, all task error messages written to standard error are redirected to a file called *ERRORS*. An *ERRORS* file with a size larger than zero indicates an error occurred during the task. After a task is executed, the *OUTPUT* and *ERRORS* files are archived together and are returned as the result.

A.5.2 An Example With Data File Caching

Here we give a slightly more advanced example, demonstrating the file caching segment. Recall that executable files are cached by default. Project wide configuration files should make use of this example to prevent clients from repeatedly downloading them:

```
data:( data1 configFile )
exec:( exec1 exec2 ):( Darwin-8.0.0-powerpc Linux-2.4-i386 )
cached:( configFile)
{exec1}:( configFile data1 ):( myoutputfile )
{exec2}:( myoutputfile ):()
```

Here we define a data file, *data1*, and a configuration file, *configFile*. We also declare two executables, *exec1* and *exec2*. Two operating systems are supported this time. The configuration file, *configFile* is declared to be cached with the **cached:()** segment. The equivalent command line is: "./exec1 configFile data1 & ./exec2 myoutputfile". In this example we explicitly declared an output file called *myoutputfile*. This is a file that we opened and wrote data to using language or system calls. The contents of *myoutputfile* are generated during the execution of *exec1,* if the return code of *exec1* permits, *exec2* is subsequently executed passing *myoutputfile* as the only command line argument.

A.5.3 A MISD or MIMD Example

Perhaps a project needs to execute each executable only once, as in a MISD or MIMD scenario. In this case, we want to override the default caching for executable files so that they are used once and then discarded. We do this using the **notCached:()** segment:

```
data:( configFile )
exec:( exec1 ):( Darwin-8.0.0-powerpc Linux-2.4-i386 )
cached:( configFile)
notCached:( exec1 )
{exec1}:( configFile ):()
```

Note that the *cached* and *notCached* segments must appear after the *data* and *exec* segments.

A.5.4 An Example with a Command Line Argument Flag

Passing flags and arguments into executables is sometimes desirable. We use double quotes to specify command line arguments, options, and flags.

```
data:( data1 )
exec:( exec1 ):( Darwin-8.0.0-powerpc Linux-2.4-i386 )
{exec1}:( “--flag1” “value” “--input-file” data1 ):()
```

This manifest is equivilent to “./exec1 --flag1 value --input-file data1”. This manifest demonstrates how to pass command line options into your executables.

A.5.5 Access to Virtual Machines, Compilers, and Interpreters

One of the design goals of SimpleDS was to ensure flexibility. It is designed to run task executables that require a virtual machine or interpreter. We use the term *stub* to refer to such a non-native subsystem. It should be noted that our primary focus in development is getting native code working, and the stub implementation is incomplete, we are presenting the manifest syntax here for completeness and to highlight the potential for flexibility.

```
data:( data1 )
stub:( java ):( 1.4):( exec1.class )
{exec1.class}:( data1 ):()
```

This manifest defines a task that runs the class *exec1.class* on the JVM version 1.4. There is a single data file, *data1*. This is equivalent to the command line “java exec1 data1”.

Perhaps we want the clients to compile the java class using *javac* version 1.4 and then execute the class on the appropriate JVM:

```
data:( data1 exec1.java )
compiler( javac ):(1.4)
stub:( java ):( 1.4):( exec1.class )
{javac}:( exec1.java ):(exec1.class )
{exec1.class}:( data1 ):()
```

Note the slight difference between the use of the **stub:()** segment and the **compiler:()** segment. The stub segment is used to declare scripts or byte code that is dependent on a virtual machine or interpreter for execution, whereas the compiler segment defines an exposed system executable that will generate executables or byte code files that can be run later or returned as a result. An example of compiling a C program and then executing it on some data:

```
data:( file1.c file2.c file3.c data1 )
compiler( gcc ):(3.3)
{gcc}:( file1.c file2.c file3.c ):(a.out)
{a.out}:(data1):()
```

Here we compile three C files into one executable, *a.out,* and then execute *a.out* to process our data file, *data1*. Notice that the supported operating systems are not defined. Any source code must be portable and account for the platform. You can see having access to compilers could be useful for MIMD projects with many small executables but probably should be excluded from use in a more general scenario.

Perl, Python, Ruby, and the Parrot virtual machine could be accessed fairly easily using our stub context. Scripting languages make heavy use of extendable modules, Perl is a good example of this. Installed modules vary from system to system, so dependent modules are encouraged to be distributed as cached data files in conjunction with the use of command line arguments to define the include directory as "."

Stubs are not automatically available, each compiler or virtual machine that a client wants to expose must be explicitly registered with SimpleDS. In the future, it may be possible to *only* expose stubs and prevent clients from executing native code. Again flexibility is a design goal.

A.6 System Management and Performance Monitoring Tools

The primary tool used for investigating projects, project files, and tasks will be *sdsshow*. This tool has not yet been implemented. *Sdsshow* will display some information about the current projects, project files, and project tasks. For example, it may show a list of all the projects along with the number of complete tasks and the total number of tasks. It may also show some statistical information, such as turnaround time, or average turnaround time per task, etc. It will also display information about the status of a task. In general, the *sdsshow* tool will be used to query the back-end database for the most common information. The interface for this tool has not been finalized. We envision something like the unix command *top*.

The tools for deleting projects, tasks, data files, and executables will be: *sdsdelproj*, *sdsdeltask*, *sdsdeldata*, *sdsdelexec*. The interface for these tools will be in the form **<tool name> <URI> <val>**, e.g. "./ sdsdelexec http://www.agentlab.net/projects/example1 exec1" indicates that we want to delete the executable named *exec1* from the project http://www.agentlab.net/projects/example1. Note that the executables for all platforms would be eliminated with this command. Finer grained control will likely be implemented also. Further, the state of any unassigned tasks depending on this executable would be changed to prevent their assignment. *Sdsdeldata* is the data file counter part to *sdsdelexec* and works in the same fashion. *Sdsdelproj* would eliminate the executables, data files, tasks, and results for an entire project. A flag will be available to prevent the destruction of results, however the default command will clean results also. The *Sdsdeltask* command deletes single tasks from the system.

VITA

Austin Royce Gilbert

Candidate for the Degree of

Master of Science

Thesis: LIGHT WEIGHT HIERARCHICAL CLUSTERING MIDDLEWARE FOR PUBLIC-RESOURCE COMPUTING

Major Field: Computer Science

Biographical:

Educational: Graduated from Marion High School, Marion, Kansas in May 1996.; received Associate of Applied Science in Computer Information Systems and Associate of Arts in Liberal Arts from Tulsa Community College in July, 1998 and July, 2002 respectively; received Bachelor of Science in Computer Science from Oklahoma State University, Tulsa, Oklahoma in August, 2002; Completed requirements for Master of Science degree with a major in Computer Science at Oklahoma State University, Stillwater, Oklahoma in May, 2007.

Experience: Employed as a *network security analyst* after completing Bachelor's and before returning to pursue a Master's degree; worked for Oklahoma State University, Department of Computer Science as a research assistant and teacher's assistant during Master's course work.

Name: Austin Royce Gilbert Date of Degree: May, 2007

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: LIGHT-WEIGHT HIERARCHICAL CLUSTERING MIDDLEWARE FOR PUBLIC-RESOURCE COMPUTING

Pages in Study: 53 Candidate for the Degree of Master of Science

Major Field: Computer Science

Abstract: The goal of this work was to investigate ways to implement and improve a public-resource computing middleware. Specifically, to make hosting a public-resource computing project logistically simpler and to examine the affect of hierarchical clustering on bandwidth utilization at the central server. To this end, we present the architecture for our cross-platform, multithreaded public-resource computing middleware.

Implementing and debugging the middleware proved far more challenging than initially anticipated. As hard as debugging multithreaded programs is, our experience has shown us that it can be leveraged to simplify system components. Our main contribution is the final system architecture.

Advisor's Approval: Dr. István Jónyer