THE UNIVERSITY OF CALGARY

Grid Computing with Plan 9 – an Alternative Solution for Grid Computing

by

Andrey A. Mirtchovski

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
May, 2005

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Grid Computing with Plan 9 – an Alternative Solution for Grid Computing" submitted by Andrey A. Mirtchovski in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Robert William John Simmonds,
Department of Computer Science
Supervisor

Dr. John Aycock,
Department of Computer Science
Internal Examiner

Dr. Ronald G. Minnich,
Advanced Computing Lab, LANL
External Examiner

Dr. Lewis Joshua Leon,
Department of Electrical and Computer
Engineering
"Internal" External Examiner

Date

Abstract

This thesis describes tools and utilities facilitating collaboration between geographically separated computer installations. The goal of this research is to allow the set up of a fully operational distributed computing environment, a testbed grid network called *9grid*. Compared to other software support for grids, the main benefits of 9grid are simplicity, scalability and tight integration with the environment it serves.

This thesis describes the implementation of several tools which aid computation across administrative domains. These tools include the hardware, software and operating system monitoring kernel driver, `devmon` and the resource discovery tool called ResFS.

Devmon is a grid-oriented hardware monitor which integrates data collected from various performance counters into a unified view of a system's current status. The devices monitored include hardware performance counters, temperature and fan sensors, operating system profiling variables and, if available, individual job performance measurements. The information provided by several devmon monitors running on several machines can be combined to provide a global view of the grid's status.

ResFS, on the other hand, is a distributed resource discovery file system for 9grid. ResFS is a hierarchical file system built on top of the 9p protocol which runs on the "Plan 9 from Bell Labs" operating system. ResFS presents a directory structure of file names corresponding to nodes on 9grid and resources available for those nodes. The architecture of ResFS is multi-tiered. Nodes on the grid announce themselves with one or more registry services, which act as aggregates of information about a subset or a complete set of nodes and resources on them that are available at any one time.

The thesis also discusses collaboration tools and cross-administrative authentication mechanisms for 9grid which are currently in development.

Acknowledgments

I would like to thank my supervisor, Rob Simmonds, for his endless enthusiasm. His fascination with distributed systems and computing has been contagious. Thanks to Ron Minnich for his support and encouragement to "think outside the box" and for showing me that the beauty of computer science lies in simplicity. Thanks to Brian Unger for bringing together some of the brightest young minds and for creating a thoroughly enjoyable research environment in the Grid Research Centre at the University of Calgary. Thanks to Nayden Markatchev, Mark Fox, Phil Rizk, Roger Curry and the rest of the Grid Research Centre group for the wonderful lunch-time discussions which not only put my work into perspective but, through the unselfish sharing of ideas so characteristic of this group, managed to keep me focused on what's important. Thanks to the Bell Labs Systems Research team for creating a wonderful operating system which made the task of writing distributed software seem almost trivial, and also for enduring my ceaseless questions. My parents and family deserve a special acknowledgment for infusing me with the hunger for knowledge and the desire to better the world through science. Lastly and most importantly, for standing by me through the two most important years of my life, and for her unconditional love, my wife, Petya.

Table of Contents

List of Figures

List of Tables

Beauty is more important in computing than anywhere else in technology
because software is so complicated.
Beauty is the ultimate defense against complexity.

– David Gelernter
(in *Machine Beauty: Elegance and the Heart of Technology*)

Chapter 1

Introduction

This thesis describes the design and development of tools that support grid computing using the *Plan 9 from Bell-Labs* operating system. The software tools described in this thesis provide hardware monitoring, resource discovery and authentication for the testbed distributed environment, *9grid*. The goal of this thesis is to provide software which allows networked systems to be more easily integrated into a distributed environment comprised of a geographically widely distributed set of users.

This chapter introduces the concepts of distributed computing and grid computing. Section 1.1 describes some of the more commonly used terminology related to grids, distributed systems and meta-computing software. Section 1.2 discusses the motivation for this research, listing the major goals that I have tried to accomplish and justifies the use of the tools that I have chosen. Section 1.3 gives a historical perspective of the evolution of distributed and grid computing. Section 1.4 details the contributions that my work offers towards creating a usable and elegant environment which integrates widely separated computing resources. Finally, section 1.5 presents an overview of the rest of the thesis' chapters.

1.1 Terminology

Connecting computers to work together on the same problem has long been adopted as a way to avoid, or share, the cost of expensive supercomputers. In fact, high performance computers comprised of many single- or dual-cpu nodes connected via fast networks occupy a significant share of the world's top five hundred fastest computers list [46]. All of the top ten machines currently (November 2004) are comprised of less powerful server- and workstation-class nodes linked together by fast networks.

The term "distributed computing" is commonly used to describe any sort of operation that involves two or more computers connected by networks. Distributed computing was a hot topic in computer science research a decade ago, however a general solution that was able to encompass all aspects of the possible arrangements of computers and networks in a distributed environment was not found and the field split into several distinct subsets, each named differently and having a distinct function, as described below.

Cluster computing involves a set of nearly identical nodes in close proximity, connected via gigabit or fiber-optical networks; such installation of identical hardware located in the same physical space is commonly referred to as a *cluster* [2]. Clusters can be built out of specialized hardware, or using readily available off-the-shelf components. Since the only hardware support for distributed computation in clusters is the network connecting the nodes, all of the application-level parallelism in such environments is achieved via software middleware such as:

- Message Passing Interface (MPI) – programming model
- Parallel Virtual Machine (PVM) – programming, execution and debugging model
- Distributed File Systems – storage

Commodity-off-the-shelf (COTS) cluster computing evolved in the early nineteen nineties. With projects such as "Beowulf" at NASA [1] in 1994 cluster computing has matured to be one of the major players in the high performance computing field. The current generation of cluster software provides high performance at low cost, propelling clusters to the top of the list of most powerful computers in the world.

Meta-computing gained popularity as an expression describing computers connected and controlled by software to perform distinct functions. Enterprise-level meta-computing software is responsible for connecting resource pools such as databases, storage and web servers and employee workstations with a corporation' data center. Essentially meta-computing provides an environment in which applications are not tied to the underlying hardware they execute their code on. Meta-computing emphasized systems tailored towards solving the problems of a single organization. It relies on centralized control and strict policy enforcement with the core of the system usually inaccessible by its user, who is only allowed access to parts of the system based on the functionality or data he or she needs. An example of such configuration is any university accounting system. The hardware involved in meta-computing usually includes large, enterprise-level multiprocessor machines, mainframes and storage servers federating the bulk of the workload and number-crunching powers into a data center.

Grid computing bridges the gap between meta-computing and clusters by describing a computational model in which jobs access a collective pool of resources from computers which may be physically separated and spanning across administrative domain boundaries [4]. A relatively new field in Computer Science, grid computing involves the utilization of widely distributed resources toward the goal of solving complex problems requiring sharing and federating resources between research organizations. Thus, the next evolutionary step of High Performance Computing will most likely involve a departure from the single all-encompassing supercomputer towards multitudes of, most likely small, heterogeneous grids of computers, connecting together the computational powers of today's clusters with other scientific resources [1][2].

1.2 Motivation

There is a trend towards building large pools of computational resources by linking together the resources of different organizations into a collective resource, shared among the participating entities. Clusters, (and recently grids) emerge as economical alternative to the single supercomputer model of computation, but they also bring a social and cultural change by connecting together people belonging to different institutions but working on the same problem in virtual organizations [4].

The work described in this thesis explores a new approach to designing distributed systems by observing that the easiest way to connect physically separated resources is to provide a common element, a sufficiently simple *glue* protocol to connect them. This thesis describes a set of middleware programs which seamlessly connect resources and services separated not only by physical distances, but also by administrative and organizational barriers. *9grid*, the testbed grid environment which the tools described here are used in, extends an existing distributed operating system, Plan 9, which has a proven track record in connecting software and hardware into a seamless environment, adding to it the most important features of grid computing: resource discovery, system monitoring and cross-administrative authentication mechanisms.

The motivation behind this research can be summarized in one word: simplicity. The goal is to create an environment in which the following are true:

- applications require minimal modification in order to be grid-enabled,
- creating a new distributed application from scratch is not complicated,
- the entry barrier is lowered significantly by allowing the end user to make the decision whether to join and share resources with others,
- the cost of setting up the environment is small,
- there is very little user-perceived *paradigm shift*, i.e., users are not required to adapt to a totally new and unfamiliar environment.

Essentially this means that the grids built with this system are supposed to be cheap, easy to use and easy to program for. The interest with which the Plan 9 community has adopted 9grid and the eagerness others have expressed in joining resources in utilizing the environment are a testament to its contribution in the field. Currently there are 9grid nodes in five different countries spanning three continents.

1.3 Historical Survey

The evolution of distributed computing can be traced back about 50 years and runs in parallel with the evolution of High Performance Computing (HPC) [2]. HPC started when the first general purpose machines were built to aid the war effort during World War II. Historically the fastest computers in the world have always fallen into two categories, supercomputers and clusters of scalar uni- and multiprocessors. In both types there exists a strong reliance in parallelism to achieve the goal of very fast computation. Since all computers have a practical peak performance, to go beyond that they need to be either upgraded with faster components and additional units, or to be clustered. The common denominator in all cases is the increase of the ability of the hardware to execute more instructions per unit of time.

In the 60's, Seymour Cray, introduced parallel instruction execution and pipelined function units, which allowed the creation of the first vector supercomputer, the Cray 1, in 1975. Several other companies soon followed in offering "vector supers": IBM, Fujitsu, Hitachi, NEC. Some of them are still in the market, building vector machines including the currently second fastest computer in the world, the "Earth Simulator", a collection (or a cluster) of vector processor units built by NEC in Japan. At the same time other companies designed and built cheaper and slower offerings nicknamed "mini-supers" [2], which appealed to universities and organizations with smaller budgets. Companies who offered "mini-supers" at that time included Alliant, Ardent and Convex.

CMOS-based microcomputers started appearing in the seventies. They brought in a trend towards moving away from large, centralized time-shared computers towards smaller, personal computers albeit at a general loss of performance. The CMOS architecture suffered from an inherent lack of parallelism – while vector processors were able to work on several chunks of data at the same time, CMOS-based microcomputers operated sequentially. Since they were much cheaper, they became the preferred building block in the movement towards personal workstations. Workstations allowed people to circumvent oversubscribed timesharing machines and have a computer dedicated only to them, even if that came at a net loss in performance. Workstations also caused a move towards a single-user emphasis on operating systems which themselves were created as timesharing environments.

The proliferation of relatively fast networks following the successful ARPANET project signified the importance of connecting those single-user computers together. Furthermore, workstation-based environments created many issues for system administrators ranging from common security and user authentication problems to software update and licensing on thousands of workstations. As Pike writes in [7], Chapter 1: “The early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems. Timesharing centralized the management and amortization of costs and resources; personal computing fractured, democratized, and ultimately amplified administrative problems. The choice of an old timesharing operating system to run those personal machines made it difficult to bind things together smoothly”.

These problems with Operating Systems of the time soon brought many efforts to create environments which utilized networks to the fullest extent. The first universally acknowledged distributed software operating system is the RSEXEC Network OS for the ARPANET, built in the early 70’s [45]. AEGIS, AMOEBA, Choices and DUNIX [41] are all examples of distributed operating systems created in the heyday of distributed computing in the late eighties and early nineties. Many old-style timesharing operating systems offered distributed computation support in one form or another as a part of the installation, such as the *r*-commands in UNIX for example. The “Plan 9 from Bell Labs” operating system also emerged at the same time. It was the result of the work of researchers from the group that created UNIX at Bell Laboratories, who were addressing the system administration and interoperability issues of running an OS designed for single timesharing computers in a networked environment.

The further proliferation of networks and workstations forced vendors to develop interoperability tools for network computing. Where before it was sufficient to provide interoperability solutions only between workstations and the central processing mainframe or the data centre, now it is necessary to be able to communicate with any number of diverse systems. The solution adopted by most developers is to create a middleware interoperability layer which offers an identical view of a computer’s services regardless of the underlying operating system.

In the late 90’s people became aware that the next evolutionary step in scientific computing must not be in the form of faster hardware, but will come as a unified mechanism for accessing resources across the organizational boundaries. The official birth of grid computing is considered to be the publication of “The Anatomy of the Grid” in 1998 [4]. It defined the major building blocks of computational grids to support the emergent *virtual organizations*. Virtual organizations are collaborative teams which do not need to belong to the same organization, but work together on the same project and thus need to share resources.

Currently grid computing middleware offerings exist from both academia and commercial entities, though most of them are not really geared towards providing pervasive access to data and computation. For example Sun’s “GridEngine” and Oracle’s “10g” database are enterprise-level solutions for data and compute centers. Avaki’s Legion Grid Portal [17] is perhaps the only “true” commercial grid offering, aiming to provide an environment for federating data, computation and authentication resources. It is designed as a portal, providing easy-to-use interfaces and uses standard, off-the-shelf software in conjunction with specially developed command-line tools. The major drawbacks of Legion, are interoperability and the reliance on a non-standard (new) set of tools and commands.

Academic research into grid computing combines open source designs, accessibility and interoperability are the preferred building blocks among university research groups. There are ongoing standardization efforts based primarily on open source, academic solutions.

The Globus toolkit [26] was created in the late 1990s as part of a joint research project between Argonne National Laboratory and the Information Sciences Institute at the University of Southern California. The team of scientists built upon NEXUS[21] in the mid 90’s to create a toolkit for securely connecting resources across networks. Nexus[20] is a communication library which supports a wide range of process creation mechanisms in clusters and in its security-enhanced version [19], across the Internet. Out of their research came the Globus Toolkit [15]. Globus provides middleware support for legacy systems thus allowing grids to take advantage of the huge base of applications software that those legacy systems support.

The aim of Globus is to provide a solution to the computational needs of large virtual organizations that span multiple institutional and administrative domains [4]. Globus is a middleware toolkit that provides fundamental distributed computing services such as authentication, job starting and resource discovery.

While there are other Grid computing toolkits available, Globus currently provides a *de facto* standard due to its development being closely tied to the formulation of the Open Grid Services Architecture (OGSA) standard [3]. Once OGSA and other standards being developed by the Global Grid Forum (GGF) are finalized, it is likely that commercial toolkits that adopt these standards, and therefore can inter-operate with Globus environments, will become more popular.

Globus provides a collection of *services* [5] including: Grid Security Infrastructure (GSI) which provides authentication based on a Certificate Authority trust model [14][25]; Grid Resource Allocation Manager (GRAM), which handles job starting or submission; extensions to the FTP standard to provide GSI authentication and high performance transfer (GridFTP) [16]; Monitoring and Discovery Service (MDS) enabling remote resource discovery.

By itself, Globus does not provide all of the tools and services required to implement a full featured distributed computing environment. Additional tools are available to fill some of the gaps. The National Center for Supercomputing Applications (NCSA) provides a patch to add GSI authentication to OpenSSH. This allows Globus environments to have terminal based single-sign-on. Globus does not provide any scheduling functionality, but rather relies on the client operating system scheduler or batch schedulers such as OpenPBS [29] to handle local scheduling activities. Global scheduling between Globus processes can be provided by meta-schedulers, such as Condor-G [30]. Condor-G submits jobs to the GRAM service running on Globus nodes and GRAM handles the task of submitting the job to the local scheduling system.

Many grid projects are already successfully operating across the world using the Globus toolkit. These projects include WestGRID [42], TeraGRID [43] and RealityGRID [44].

The problems which people experience with those toolkits come primarily from the fact that the underlying operating systems lack support for distributed computing. Most toolkits implement the basic connection mechanism themselves, often resulting in incompatible or, alternatively, tightly coupled, technologies and the consequent software bloat. For example, the Globus Toolkit v3.2 consumes, in archived form, twice the space of the entire Plan 9 operating system. The existing middleware toolkits have a set of administrative requirements which need to be satisfied before even the very modest demands application is to be executed, which is a problem when it comes to writing new code or modifying an existing one. The expression "grid-enabling an application" is used to describe the process of rewriting an application to work with a particular grid toolkit. Infrastructure, installation and maintenance are hindered by the reliance on custom-patched libraries and lack of documentation.

Even though major standardization efforts are ongoing, few standards exist, and in the cases where there are standards they are either not enforced or continuously evolving [54][56][57]. Furthermore, there is an observed trend for users to avoid the tools provided by the middleware toolkits and instead use ones that they are more familiar with from their day-to-day computer use. For example, file transfer statistics on WestGrid's networks (WestGrid is a grid computing network spanning across several western Canadian universities [42]) indicates that the majority of file transfer traffic goes through SCP, the SSH-based encrypted file copy program, instead of the GridFTP program provided by the Globus Toolkit. Naturally this percentage will change once people familiarize themselves with Globus' tools but it is nonetheless indicative of reluctance among users to adopt new technology, even when the potential benefits outweigh the drawbacks incurred from climbing the learning curve.

1.4 Outline of Contributions

The set of tools described in this thesis are a part of a project to connect many of the Plan 9 researchers, organizations and enthusiasts into a collaborating environment where they can share computational and other resources. The environment is called *9grid* [59] and currently involves researchers from the University of Calgary, Los Alamos National Laboratory, Bell Labs, Monash University and countries such as Finland, the Netherlands, Russia, Bulgaria, Japan, the UK and the US. The software support for 9grid I describe here is the first step in building the environment.

I attempt to solve some of the problems which plague middleware toolkits through integrating a small set of tools onto an already existing framework for distributed computation. 9grid, the research grid built using these tools, is substantially more lightweight, modular and portable across hardware platforms. With the new set of Plan 9 support libraries for UNIX [58] it is even portable across operating systems.

With the relative obscurity that the Plan 9 operating system has in a research environment currently dominated by Linux, there is little chance that many users will start using and building distributed environments based on Plan 9 grids. There is a possibility however, and some people already warn of this, that Linux and other UNIX clones have reached a critical mass in which the sheer size of their code base stifles non-incremental and innovative research. If this happens I expect an influx of researchers looking for simpler, more elegant systems that don't carry as much baggage left by legacy software. One of those systems may be Plan 9.

There already is a push by a few people in several universities and research labs for work on alternative operating systems, including highly decoupled service-based systems for embedded devices [34]. If such research gathers momentum this thesis will increase in value.

The contributions of this thesis include a comprehensive description of the application of Plan 9 from Bell-Labs and the tools it provides in the context of grid computing. It offers a distinct look into alternative solutions for distributed computation by breaking the mold of legacy software and building on top of systems which have been designed to accommodate new paradigms such as single-protocol communication, virtualization and a generalized approach to resource sharing.

This thesis also describes a model of integration which creates a distributed environment based on a single, well-defined protocol with which all parts of the system communicate, as opposed to the model of building decoupled service/client models in which every service defines its own communication protocol. Simplifying distributed environments is also a topic this thesis is concerned with. Tools on which we depend for our daily work are made available in a new, distributed context without modification or change in their appearance; the programming model attracts with its simplicity. My goal is to create an environment where connecting components is easy, an environment where resources share data across networks as seamlessly as they do it across pipes. This view of grids as a workflow management system akin to the pipes in UNIX is described in greater detail in Section 5.3.

Last, but not least, I hope the work in this thesis contributes to the community a set of tools which are general enough to be useful in daily system operations or even in environment which can not be described as grids.

1.5 Thesis Overview

This thesis is organized as follows:

Chapter 1 (this chapter) introduces the subject of grid computing, outlining the goals of this research, its historical perspective, motivation and contribution to the field of Computer Science.

Chapter 2 discusses the operating system which serves as the base for 9grid. It explains its basic features as they pertain to grid computing, such as:

- Private Name spaces – an OS paradigm aiding job and service virtualization,
- The *9P* protocol – the language spoken between all services on 9grid.
- The security mechanisms ensuring secrecy of communication,
- The authentication mechanisms regulating access control.

Chapter 3 deals with the tools developed for this research allowing us to monitor the hardware status of a machine on the grid. It details a kernel devices called a *monitor*, or *mon* for short and the statistics it gathers from temperature sensors, fans on the motherboard and kernel profiling variables which may be used in failure prediction or in making resource scheduling decisions.

The `devmon` kernel driver also serves as an example of the simplicity with which grid services can be created using the distributed tools of the Plan 9 operating system. Despite the fact that it is a kernel device, `devmon` can also be used as a grid service by remote clients.

Chapter 4 describes a resource discovery file system developed for 9grid named *ResFS*.

ResFS is a multi-tiered application which gives a single point of access to the resources available on 9grid. It is divided two parts: resource gathering at the local level (node) and resource gathering for a group of nodes.

The single node presents a directory named after itself, containing files corresponding to various parts of the system, which provide information about the current status of that subsystem when queried. The global view is a directory hierarchy of mounted single node directories.

Chapter 5 summarizes the work performed in the thesis and lays out some of the future directions this research may take. The future directions for this research are:

event-driven system monitoring – a much more scalable model for information gathering which could be used in large scale HPC, in clusters and large grids interconnected by fast networks

workflow scheduler – a resource management system which transports data across grid services and components much the same way UNIX pipes do between programs on a single machine

distributed `/proc` – a single control point for processes and jobs running on separate nodes on the grid or a cluster

distributed authentication – a decentralized authentication mechanism for grids

data caching – a mechanism to allow the job to choose which data caching/compression method is used for its communication

The remaining part of the thesis includes appendices containing source code for the programs developed.

1.6 Summary

This chapter gave an introduction to this thesis and the topics of grid computing it covers. It started by describing the common terminology used currently in the field of distributed computing such as grids, meta-computing, virtual organizations, clusters, and others. Next it discussed the major motivation behind this work, which is to create a simple and easy to use environment in which people and computers distributed across the globe can share and federate data, computational and other resources.

This chapter also gave a historical survey of the field of distributed computing, starting with its predecessors, the massively parallel supercomputers of the past, covering minicomputers and networks connecting them, the distributed operating systems that used them and the middleware toolkits spawned by the fact that legacy systems such as UNIX needed to be accommodated in networked environments. On the software side of things this chapter looked at other solutions that allow resource sharing on large scale, commercial meta-computing offerings as well as the currently preferred tool for academic research in distributed computing – the Globus Toolkit. This chapter also gave an outline of contributions, which listed how the grid environment built as part of this research, 9grid, will benefit from this work and how it may benefit the field in the future. An overview of the rest of the thesis was also provided.

Chapter 2

Plan 9: a Grid Operating System

This chapter introduces the *Plan 9 from Bell-Labs* operating system. Plan 9 is the development platform for all tools written as part of this thesis.[1] Section 2.1 gives a historical perspective of the design of the OS as it relates to the research trends in computer science at the time and also describes its design. Section 2.2 introduces the concept of private namespaces and how they benefit distributed and grid computing. Section 2.3 describes the basic building block of all Plan 9 installations, the *9P* communication protocol. Section 2.4 discusses data encryption, virtualization and private namespaces as tools for enhancing security in grids, while section 2.5 discusses the requirements for authenticating across multiple domains on the grid. Section 2.6 describes some additions to the Plan 9 operating system which help its adoption in grid computing, including a new authentication mechanism currently being developed in collaboration with Bell Labs and Los Alamos National Laboratory, which allows processes and jobs to cross administrative domains painlessly. It also describes *ChatFS*, a real-time communication service I have programmed for 9grid which, although not essential to grid computing, aids the communication between 9grid's administrators and users.

2.1 Plan 9 – the OS

Plan 9 from Bell-Labs [7] is a distributed operating system created in the late eighties; the first Plan 9 distribution was released in 1991. Plan 9 was created as an attempt to address fundamental issues with UNIX's design, arising from the fact that it was originally conceived in an environment lacking many components which we take for granted today, such as networking or graphical display terminals. Both networking and graphics were added later on in the life of UNIX, and every new layer accommodating a new type of hardware or usage paradigm has created more complexity due to the large amounts of code associated with it, and the constantly changing interface requirements. Plan 9 presents

[1] Plan 9 source code listed in this thesis is released under the Lucent Open Source license, `http://plan9.bell-labs.com/plan9dist/license`.

an environment free of legacy designs, where tools and services are built to address the best way of solving a problem, instead of concern for interoperability with legacy systems.

2.1.1 Historical Perspective

Several other distributed operating systems were created in the heyday of distributed systems research. Examples of such systems are Sprite [31] and Amoeba [32]. These systems built completely self-sustained environments by departing significantly from the UNIX model prevalent at the time. In Amoeba for example, communication with external services was difficult or even impossible, in most cases necessitating a rewrite of the communication code for the new system. The lack of developers, the very small range of supported hardware and the small user base, even compared to Plan 9, have also significantly slowed the adoption of those systems outside of their research communities. In retrospect, Plan 9 was the only research distributed OS from the late eighties and early nineties which managed to attract developers and be used in commercial projects long enough to warrant its survival to this day. In Plan 9 the ANSI-Posix environment [6] solves the portability problem and most non-graphical UNIX applications can easily be compiled for Plan 9. In most cases rewriting an application to run natively on Plan 9 involves removing large portions of code dealing with inconsistencies between UNIX variants, which simply do not exist in Plan 9.

As issues in UNIX were addressed in Plan 9's design, the system was also extended beyond the timesharing mainframe and personal workstation environment to accommodate networks and the fast growing cheap commodity hardware market. The early trend of the 1990's was to switch away from the organizational mainframe and into cheaper workstation-based computing, thus Plan 9 was designed to create a distributed environment comprised of cheap computers as terminals and fast, expensive servers as storage and CPU cycle providers [7].

2.1.2 Distributed Features

What separates Plan 9 from other distributed systems is the ease with which design considerations for new computing models were accommodated together with widely adopted and accepted UNIX paradigms. For example resources in Plan 9, be it disk storage, devices, graphical, network and other subsystems, are represented as files and directories containing files, all comprising a hierarchical structure called a *namespace* bound together with a simple communication protocol, *9P*. A uniform access protocol means that in contrast to Linux, which has almost 300 system calls to manage many different types of resources, Plan 9 has 40 system calls and a uniform method for enumerating and controlling resources.

Resource control is accomplished via a simple and consistent interface. Each device or file system presents at least two files for process interaction, one for control messages and one for data, usually called `ctl` and `data`. Control operations on the device are performed by writing plain text messages to the control file, status is obtained by reading from the same file. Any data that needs to be transferred to the device is written to the data file, any data that needs to be transferred from the device is read from the data file. This interface extends to all resources in the system and is universally adopted throughout the environment. It also works transparently over a network, as do all Plan 9 operations. For example a resource or a device such as network interface on a remote computer, can be imported and controlled as though it was local. Resources are functionally equivalent regardless of their location on the network or in a grid such as 9grid.

Plan 9 separates hardware resources on the local network based on the role they take in the environment. For example it separates CPU (handling the computation), terminal (providing interface to communicate with the user) and storage servers into networked components so that they can run on different hardware, preferably one that is optimized for the task it performs. For example it is common for Plan 9 networks to be built around multiprocessor servers as CPU nodes and have storage on servers connected to large disks via a RAID controller, with slow, but cheap computers connected to large displays as terminals. This greatly reduces the cost of hardware and maintenance since it concentrates expensive equipment in the server room, while users may run on inexpensive terminals comprised of just a display, mouse and keyboard.

A typical Plan 9 configuration consists of diskless workstations, Personal Digital Assistants (PDA), CPU servers and file servers, interconnected by wired and wireless Ethernet [7]. To the casual user the entire installation appears as a single, centralized system. Plan 9 allows users to configure their computers to run interactive programs locally and compute-intensive ones on one of the more powerful CPU servers. The protocols used to access file servers are security hardened to the extent that servers need not hide behind firewalls. The main Plan 9 file server at Bell Labs, for example, is outside the Bell Labs firewall.

Benefits of this design include the fact that it extends over the networks via rpc-like programs which serve as mount devices transmitting the 9P protocol messages to and from clients and the servers hosting the resources they access. The 9P protocol that connects applications to file systems was designed to run over networks as well as pipes or any other connections providing ordered packet delivery. As a result, it is quite simple to set up a diskless workstation with 9P connections to remote file systems.

2.2 Private Namespaces

The different services provided by the Plan 9 operating system are joined together as needed in a single namespace, private to the process which created it [13]. A Plan 9 process can pass its namespace on to its children unmodified, and even export it for descendants running on other nodes.

Just like in UNIX, the namespace of a Plan 9 process is composed of the trees of all the different file systems mounted by the kernel that is running the process. Unlike UNIX, however, which maintains only a single namespace per machine, Plan 9 can maintain a large number of namespaces which are either individual to a process or are shared by its parents and descendants. The `mount` operation in Plan 9 takes a file descriptor representing a connection to a file system and attaches it to an arbitrary point of the process' namespace. All operations on files within the hierarchy served by the file descriptor are conducted through an encrypted communication channel to the mounted file server.

Plan 9 also allows file systems to be mounted at the same point, thus creating the so called *union mount*. Ordering of lookup results in union directories is determined by controls in the mount driver – one may choose to mount a system *before*, or in front of any other systems currently mounted, *after*, in which case it will be the last result returned, or *instead*, in which case a traditional mount operation is carried out. The paper "Programming Distributed Applications using Plan 9 from Bell-Labs" [9] contains an example of file servers and namespaces used currently in Plan 9's process control and windowing system.

The namespace that a process sees is the result of a sequence of mount and bind operations. By repeating this sequence on another host with a little interpretation, the same namespace can be constructed there. This is done when a user starts a shell on the CPU server giving the user the same environment on the CPU server as on the terminal. The terminal's operating system actually exports its file systems to the user on the CPU server so that programs running on the CPU server continue to have access to devices on the terminal. Starting a graphical application on a remote server displays on the local terminal seamlessly as it writes all graphical operations to the same files in `/dev/draw` imported on the remote machine when the connection is initialized.

2.3 The 9P protocol

The *9P* protocol provides the glue between Plan 9 processes and Plan 9 servers [40]. A *session* between a server and its clients consists of requests by the clients to navigate the server's hierarchy and responses from the server to those requests. The client's requests are called *T-messages*, the server responses are called *R-messages*. A 9P *transaction* is the combined act of transmitting a request of particular type by the client and receiving a reply from the server.

Each 9P message contains a sequence of bytes representing the size of the message, the type, the tag (transaction id), control fields depending of the message type and a UTF-8 encoded payload. Most T-messages contain a a 32-bit unsigned integer called *fid*, used by the client to identify the "current file" on the server, i.e. the last file accesses by the client. Clients choose the tag for each message which is used in the server reply and ensure that no two outstanding messages on the same connection have the same tag. Multiple T-messages can be sent by the client out-of-order. The different 9P message types [40] and their role in the communication are listed in Table 2.1.

The Plan 9 implementation of the 9P protocol is written in C, but there are no restrictions placed on the choice of the programming language. Implementations of the 9P protocol have been written in Limbo (the language of the Inferno operating system) Python, Haskell and most recently Java.

Plan 9 commands do not usually see the 9P protocol directly. Instead, their `read()`, `write()`, `open()` and other calls are translated into 9P messages by the underlying mount driver.

2.4 Security

In the context of grid computing, a secure environment is one that is able to protect the communication between jobs on the system from third party observers, protect jobs from adverse effects caused by other jobs' actions, and protect jobs from resource starvation, which can be caused by denial of service (DoS) attacks, or illegal resource utilization by other users. Plan 9 does not rely on third party additions to handle communications with hardware or between nodes. It has been designed to enforce a strict security policy to which all programs must adhere [12].

Key elements of the security infrastructure in Plan 9 are the lack of *superuser* account and encryption of all communication via a ticket-based protocol using authentication mechanisms independent of the session or environment. The OS also provides encrypted data storage as a service. Furthermore, in Plan 9 all processes operate in private namespaces which hide a process' communication channels from others. Private namespaces were explained in Section 2.2.

The kernel delegates most of the security infrastructure considerations to the interprocess communication drivers such as `mount`, `bind` and the protocol carrying all information across the system, *9P*. This means that cluster and grid jobs do not have to be concerned with the underlying security infrastructure as long as they can ensure on their end that their private keys are not compromised.

2.4.1 Use of Private Namespaces

The most important security feature Plan 9 brings to grid computing is the concept of *private namespaces*, discussed in section 2.2. Private namespaces ensure that communication between processes or clients is restricted only to the parties involved and is invisible to others. Each process on Plan 9 sees a private view of the underlying system comprised of different resources' file servers bound together in a tree-like hierarchy called a namespace.

The primary security feature of private namespaces is to restrict other clients from snooping over private communication channels or even knowing that they exist. User-level mounts of remote systems in a process' namespace ensure that remote grid-enabled resources can be brought in on demand and invisibly from third parties. It is even possible to restrict the knowledge of which processes are currently running on a system by changing the permissions of the process file server `/proc`.

In Plan 9 processes share their parent's namespace by default unless an argument has been given to `fork()` to create a copy of the namespace. Clients willing to share parts of their namespace with other processes or users on the same system can do so by posting a file descriptor pointing to the root of that namespace to a special directory, `/srv`, which acts as a bulletin board for file descriptors. There is no restriction on what may be shared, except the restrictions imposed by file permissions associated with the exported files. For example, nobody will be able to access a resource exported without any read or write permission bits set.

2.4.2 Virtualization

Grid environments are often considered insecure due to their highly distributed nature, the fact that they involve different hardware and software platforms and the lack of a single system administration authority.

Grid solutions, such as Globus, being a middleware platform and not a standalone system, provide security features which deal primarily with proving a user's identity and encrypting the connection between hosts, not with protecting the general integrity of the system [25]. One particular issue that needs to be addressed in grid middleware toolkits is protecting jobs from a breach in the security of the underlying system. For example, a user submitting Globus jobs to a cluster cannot be guaranteed that the computational result isn't compromised or erroneous, or that the intellectual property of the work is preserved. Similarly, a computer connected to 9grid could have its kernel booted in debugging mode or hacked to log data off-site, thus monitoring all communication with endpoints at this computer unencrypted. There is no mechanism to restrict interaction between processes on the same installation. Some distributed installations, especially in security-conscious fields, impose the requirement that jobs running on the same cluster, though able to perform high-bandwidth communications between each other, have that communication protected from third party viewers.

A solution to this problem often employed in legacy systems are virtual environments, tools that virtualize the hardware causing each program or group of programs to run on the hardware alone. Such tools include VMWare [62] and Xen [61]. Using virtual environments restricts communication with other processes and secures the system, but has performance implications that are difficult to predict. There have been implementations of private namespaces by means of virtualization of the host OS [27], whether through support for private namespaces in the kernel [11], or through third-party solutions like VMWare, where a client's session on a particular node is started as a virtual machine containing resources available only to the user.

Sand-boxing approaches, including an implementation of private namespaces for various operating systems on which Globus runs [11] have generally failed to yield a secure system due to them being constrained to the particular node they run on, i.e. the privacy is lost once other systems are involved. The fact that there is no standard implementation of sand-boxes that would work in a heterogeneous environment also hinders their adoption as a security solution.

2.5 Authentication

My work related to authentication mechanisms in a grid environment is concerned with the ability of the environment management system to handle proving a user's identity and the deployment and storage of security information. This section presents the authentication mechanisms currently in use by Plan 9 and expands on possible ways of extending accounts on the grid to carry information relevant to an environment spanning multiple administrative domains.

Authentication in the Plan 9 distributed environment [12] is delegated outside of the application and is performed by dedicated authentication, or *auth* servers. Authentication, or "auth" servers on a Plan 9 network are physically secured machines that operate independently from the rest of the installation. Access to auth servers is normally restricted to the physical console of the machine.

The authentication agent in Plan 9 is called *factotum*. Factotum is the only program in Plan 9 that understands authentication protocols, security keys and the mechanisms for their deployment. This agent stores the authentication keys for the programs with which it shares the environment and performs authentication both as a client and as a server, being able to assume both sides of an identity proof dialog in the course of a single session.

Factotum does not communicate with external programs directly, instead it is *consulted* by local entities sharing its namespace whenever authentication information is needed. After receiving a request for establishing a trust relationship, a client will act as a proxy, relaying communication messages between the client's and server's factotums until a mutual authentication is reached.

Factotum also serves a double-purpose as a *single sign-on* agent with its ability to remember the currently active authentication tokens for the private namespace it serves. Since factotum stores user-provided authentication information, such as VNC or SSH, POP3 and IMAP passwords [12], instead of prompting the user on each authentication, it simply uses the tokens it already holds, prompting only when they fail. Failure is most likely to happen with one-time password schemes such as crypto cards or netkey. In effect, this creates an environment where passwords are not prompted for more than once, and once authenticated with, remote servers remember the job's owner identity without compromising the integrity of the underlying protocols.

Factotum protects the security of the account it serves by holding all security keys in protected areas of main memory, making it invisible for other users on the system. It also protects its running process' image by disabling a kernel's ability to swap it out to secondary storage. Factotum is unable to keep state between restarts because all keys are kept in volatile memory cleared before starting each new Plan 9 process. To initialize it a user either supplies the passwords requested, or bootstraps factotum on start-up by reading the keys from a general-purpose encrypted data storage called *secstore* [12], where access is controlled by an authentication mechanism separate from the system.

The security protocols currently in use by Plan 9 include X.509 certificates, RSA keys for use with SSH, the DES challenge-response protocol, and some plain-text password schemes such as the ones used by Telnet, FTP, POP3, IMAP and VNC among others [12].

The proliferation of distributed and grid-like environments, each having multiple sites and numerous authentication domains, requires a reevaluation of the currently used authentication mechanisms. Normally authentication is based on a name and unique number designating a particular user on the system, a scheme which often runs into problems even in locally distributed environments due to the special care that needs to be taken to synchronize the username/userid pairs between different systems. Plan 9 has avoided that issue by not using a numerical user ID. However, the user name of a participant is not sufficient to provide all necessary information about the user's identity in a distributed environment any more. Administrators must deal with potential user name clashes between grid sites early on in the environment's design phase.

The scheme currently used by other systems, such as Globus, creates a global user namespace, where all necessary user information is carried in their authentication information. Local grid nodes then choose their own mapping from the global user ID to a local user ID, of which there is a particular set and naming convention created beforehand to be used by Globus patrons. For example jobs started by user `andrey` at a local terminal run as user `b102` at site `B` and user `c374` at site `C`. The conversion between users is done transparently by GridFTP [16] while copying the files. Note that in the Globus scheme, all applications which might bump up against the global name/local name dichotomy need to be modified to accommodate it, the current policy for WestGrid for example is implemented using a global user name for all sites [35].

2.6 Additional Provisions for Grids

Grids require much more than simple resource monitoring, resource discovery and scheduling mechanisms to be actively used and supported. Here I discuss two tools for Plan 9 which I have taken an active role in designing or implementing. One is a decentralized cross-administrative authentication mechanism, the other is a groupware and communication service.

2.6.1 Authentication in a Grid Environment

I am currently taking an active role in investigating a new system for identifying users' memberships in organizations and locations across 9grid without necessarily storing all user information on all sites [34]. It involves designating users as members of authentication domains, where membership is carried by the user's identification name as it appears on different grid systems. For example, a process owned by user `andrey`, member of the `ucalgary` authentication and administrative domain running on a remote grid node will appear there as user `andrey@ucalgary` and will be unable to access resources which a member of the remote administrative domain with the same user name may have access to.

Local authentication servers are able to request credentials from the master auth servers for a member of a particular domain without having to store them locally. They can also refuse authentication to untrusted members, even though their credentials with the remote server may be valid. This is implemented by extending the authentication server's capabilities to include authentication via remote systems and auth servers not handled by the immediate network being authenticated for via authentication proxies.

This authentication scheme aids grids by allowing users to keep their desired user names across administrative domains, while avoiding clashes with local users' processes and providing administrators with an easy way of identifying where a particular process originates from. The authentication server the user is assigned to in this case acts as a global authentication agent for this user's processes, authenticating their identity everywhere on the grid. It also simplifies administration by keeping all administration-related tasks close to the originating site, instead of delegating them to a centralized administrative entity.

This scheme is very similar to the REALMs implemented in Kerberos V4 and V5 [67]. The differences are mostly in the details: where the Plan 9 scheme described above requires no additional tools and programs to operate and be administered, Kerberos requires extra software to be installed on the system, such as `ksu` or `kadmin`, which are used for administrative tasks. The configuration file for cross-realm authentication needs to exist on all clients in all domains involved. There are no mechanisms for excluding domains from cross-realm authentication, unless a new realm is created containing only those systems.

2.6.2 ChatFS: A Groupware Communication Service

A grid is useless if it doesn't provide a way for its users to communicate with each other and synchronize on their work, which is why I have developed a file-server-based chat program called *ChatFS*.

Chatfs is a service which can either be used in a grid environment or in the context of a single Plan 9 installation. It works by providing a file descriptor which, when mounted by its clients, serves as the root of a directory in which several files corresponding to "rooms" are located. Clients who wish to join a particular room cd to the directory and run a simple two-way communication program that reads and writes to the file corresponding to the room.

Chatfs takes care to announce each new client, by printing a short message indicating the user name of the person who joined. It also ensures that each message written by its clients is transmitted to everybody else in the room.

The differences between ChatFS and other networks providing similar services such as Internet Relay Chat (IRC) [68] or Instant Messaging is that ChatFS is integrated with 9grid and does not require any specialized protocol or clients. Chatfs has been in use for a few months on the University of Calgary's Plan 9 installation with clients connecting from all over the world.

2.6.3 Other Considerations

Other programs which I am taking an active role in designing, and which will become part of the Plan 9 grid toolkit, include:

- distributed `/proc` file system – the ability to control processes executing on remote systems

- meta scheduling – decision making about the location a program executes at based on available and demanded resources and policy
- distributed data replication – a unified framework for managing data on the grid
- data transfer and caching – how to transfer data fast and how to cache data during and after computations runs

Those additions are listed in detail in Chapter 5, Future Work.

2.7 Summary

This chapter introduced the *Plan 9 from Bell Labs* operating system and the distributed computing features it provides. The chapter discussed the historical perspective of the OS and how design decisions affected features such as security, authentication or remote service access. It introduced the concept of private namespaces, perhaps the most important feature of the OS, and gave examples of how private namespaces help in securing the Plan 9 environment. The chapter also discussed the single sign-on authentication agent *factotum* and the help it provides in generating a secure environment without the hassle of supplying passwords at each step. The chapter also described 9P, the communication protocol on which all Plan 9 services are based, both local and remote.

Finally, the chapter discussed some additions to the Plan 9 operating system to help its adoption in grid computing, such as a new authentication mechanism that crosses administrative domains, a groupware communication service called chatfs and several other programs still in development which I have taken an active role in designing.

9P message type	Description
version	identifies the version of the protocol and indicates the maximum message size the system is prepared to handle
auth	exchanges auth messages to establish an authentication fid used by the attach message
error	indicates that a request (T-message) failed and specifies the reason for the failure
flush	aborts all outstanding requests
attach	initiates a connection to the server
walk	causes the server to change the current file associated with a fid
open	opens a file
create	creates a new file
read	reads from a file
write	writes to a file
clunk	frees a fid that is no longer needed
remove	deletes a file
stat	retrieves information about a file
wstat	modifies information about the file

Table 2.1: Message types in the 9P protocol

Chapter 3

System Monitoring

Today's Grids are an amalgam of resources distributed across networks. To gain access to those resources and to be able to utilize them a grid client needs to know what is available at any given point and where it is located on the network. It is a common case for grid jobs to request a specific hardware type, system load, storage or network capacity. The role of a grid *system monitoring* tool is to provide easy access to that information.

System monitoring makes available information regarding the state of a particular computer, its operating system, jobs running on it and the network connections made to or from it. System monitoring also includes information about hardware devices such as storage drives, network interfaces or any secondary devices that may be present, such as audio or USB devices. Some architectures, such as IBM's POWER5, have specialized performance counters which also play a role in performance monitoring. They provide valuable information about how the system behaves under different workloads.

This chapter describes *devmon*, a Plan 9 kernel driver I have developed which presents system monitoring information as a network-accessible file hierarchy. Devmon is used in conjunction with other tools to aid monitoring, development and maintenance of the 9grid research grid [59][52].

The chapter starts by describing the role of system monitoring in grid computing. It gives justification for the need of a robust architecture providing an interface to a system monitoring information for a widely distributed set of computers. The design and features of devmon discussed in Section 3.2, include the data format, establishing connections to devmon and accessing its information and control interface. Sections 3.3 and 3.4 describe the hardware and OS monitoring features currently implemented in devmon. Section 3.5 presents performance measurements and evaluation of devmon, while 3.6 compares it to other system monitoring tools. Section 3.7 talks about the future development planned for devmon.

3.1 System monitoring in Grid Computing

In the context of grid computing, system monitoring expands to include information about all available computers and resources on the grid, a task much more difficult than that of monitoring a single system. System monitoring coupled with resource discovery (discussed in Chapter 4) provide the information required to make decisions about sharing or scheduling grid resources. In the short term system monitoring can help observe and evaluate critical conditions on the grid; in the long term it helps observe trends by providing information which can be statistically analyzed.

For example, by knowing that a set of nodes is currently busy with a CPU-intensive job a resource scheduling algorithm may decide to delay the submission of a new job or to run it on different computers. Tests of the devmon monitoring software described in this chapter show that with a Pentium IV processor with core frequency of 2.66 GHz running computationally intensive jobs the mean time between a CPU fan failure and the core temperature increasing above the manufacturer's operational limit is 5 minutes. Ample time, in most cases, to prevent a disaster.

In the long run that same scheduler may observe patterns in the system's behaviour under heavy load and decide to schedule batches of potentially long-running jobs during off-peak times, such as nights and weekends. If system monitoring information is additionally saved in a database for later retrieval and review, one is able to analyze it statistically in order to discover correlations between usage patterns, which may help improve performance of the computer in question, or for a particular type of jobs running on it. Another example relevant to grids is to increase bandwidth between nodes whenever a potentially large data transfer occurs. This type of system monitoring and scheduling is used in some optical switches which are able to change packet routing policies on-the-fly and utilize high-bandwidth interfaces whenever the network load increases beyond a certain threshold [60].

System monitoring is also used in failure prediction for large HPC clusters [36], where an abnormal reading of one of the sensors on a computer provides a warning about potential disruption in the workflow or a failure of the node as a whole. Thus, observing that a CPU fan's speed has reached 0 or is falling below a low watermark one can predict an increase in the CPU core temperature and take evasive actions such as turning the node off or turning the CPU off on architectures where that hardware feature is available. Replacing a faulty fan on a CPU is much less expensive than replacing a motherboard with a melted processor on it.

Currently, hardware monitoring in Plan 9 suffers from several issues that prevent it from being fully utilized in an environment such as 9grid, which devmon is designed to solve. One is the lack of centralization in reporting the system resources available, the other is the lack of integration between system monitoring (devmon), network database services providing query services for resources (ndb) and the networked environment spanning multiples of computers from different Plan 9 installations. *ResFS*, described in chapter 4, was created to solve that problem by binding those components together and providing an easy access to information about resources on 9grid.

3.2 Design and Features of Devmon

The name *devmon* follows the standard Plan 9 naming conventions, combining the function of the device with the position it takes in the kernel. `Dev` is the common name for device drivers and `mon` is borrowed from SuperMon.

The single-character Unicode kernel-name used to attach to the device when the first connection is initiated is "∞", a name chosen randomly from the non-ASCII subset of the Unicode character set. The Unicode character set was invented by the creators of *Plan 9 from Bell-Labs* and is used throughout the system [10].

As with any other Plan 9 kernel device, devmon presents its information and control interface as files in a hierarchical directory [7]. Devmon serves a single-level directory, currently containing four files:

```
monctl
mondata
```

```
    fans
    temp
```

Two files function as control and data interface to devmon: `monctl` and `mondata`. Devmon also provides an interface to the on-board hardware monitoring chipset, `fans` and `temp`, using a slightly different format conforming to the accepted one by the standard visualization tool, `stats`. The change in format for the two files is discussed in section 3.2.3.

By accessing the files via standard tools provided by the operating system and familiar to most users such as `cat`, `ls`, `grep` and others, one can obtain information about the operating system's current status, the status of various hardware sensors or, on some architectures, performance counters.

3.2.1 Data Format

The data format accepted and output by devmon is S-expressions. This format has been chosen for two reasons: it is easy to read and simple to parse.

Symbolic expressions, or S-expressions [38], comprise a LISP-like recursively-defined data representation model used for complex, structured data. Their main benefit is that they do not require the presence of additional meta-data describing the structure. An S-expression is a list of either atoms or S-expressions: `(a (b c))`. In this example the expression contains an atom "a" and an S-expression, which contains in turn two atoms, "b" and "c". S-expressions are simple, useful, and well understood. [36]

Simplicity and readability of the ASCII-compatible UTF-8-encoded S-expressions used in conjunction with plain-text data in devmon ensures that most of the time allocated for this research was spent working on important code instead of designing, debugging and modifying XML schemas, parsers or data. A more detailed discussion on s-expressions and XML can be found on the SuperMon web site at [37] or the paper “Supermon: A High-Speed Cluster Monitoring System” [36].

Note that some of devmon's files output their information as tab-separated plain text files. This has been done to allow easier integration with existing Plan 9 tools such as stats, a data visualization program for system monitoring. Figure 3.3 shows an example of stats displaying information gathered by devmon.

I want to emphasize that the choice of data representation or the format of control messages was made out of concern for simplicity. While there are some benefits in choosing a simple set of messages formatted as s-expressions, a more ambitious project may freely switch to XML, indentation-based, binary or any other format available. Simplicity and, in the case of montcl's control messages, syntactic clarity were the main reasons for choosing the set described here.

3.2.2 Connecting to Devmon

Connection to the devmon kernel device is established with a mount system call invoked via the bind command, described in more depth in section 2.2. The device name used as an argument to bind is the UTF-8 rune which the device registers itself under "∞" in the case of devmon. The command bind requires at least two parameters: the name of the kernel device and a directory name where this device is to be bound. The -c argument in the following example specifies that writes to the kernel device will be allowed, though still subject to the permissions the device specifies:

```
% bind -c '#∞' /tmp
% cd /tmp
% ls -l
--r--r--r-- ∞ 0 bootes bootes 0 Aug 19 10:22 /tmp/fans
--rw-rw-r-- ∞ 0 bootes bootes 0 Aug 19 10:22 /tmp/monctl
--r--r--r-- ∞ 0 bootes bootes 0 Aug 19 10:22 /tmp/mondata
--r--r--r-- ∞ 0 bootes bootes 0 Aug 19 10:22 /tmp/temp
%
```

File sizes reported by devmon are given as zero, and their creation time is the start time of the program. The file size is a consequence of the fact that all files are generated on the spot and do not consume any storage space. The owner of the device is the host owner of the computer, equivalent to a reduced-privilege administrator in the Plan 9 world. All permissions are specified and controlled explicitly by devmon, with the following restrictions:

- only the host owner can write to the control file
- anyone can read from the data files
- writing to data files is forbidden

The host owner can modify a file's permissions using the chmod command. Writing to any of the data files makes no sense in the context of devmon because it won't allow any of the kernel variables to be set through it, as it would constitute a security breach. In the following example a normal user attempts to write a control message to monctl:

```
% echo 'context false' > monctl
monctl: rc: can't open: permission denied
%
```

Note that access to devmon is not limited to the local computer since the 9P protocol allows access to the files served by devmon from other computers on the network, provided that one can authenticate and connect to the original host. This simple arrangement is utilized in *ResFS*, discussed in chapter 4, where directory hierarchies served by different computers on the network are bound together to create a single view of 9grid. Thus, one can import devmon devices from various machines and query all of them from the local prompt without having to explicitly connect for each operation.

3.2.3 Accessing Devmon

Reading from devmon's files returns information about the state of the computer or the kernel module itself, which is dynamically generated. Writing to the control file a predefined string of monctl control messages will control device parameters.

Reading mondata will output an s-expression containing the name of the computer and the current values for several kernel variables which monitor performance and load on the system. It will also query the hardware monitor on computers which have motherboards with supported chipsets. An example s-expression obtained from mondata follows. Note that data has been formatted to fit the screen since pretty-printing s-expressions is not required when parsed by computers:

```
% cat mondata
(
    (sysname plan9-2)
    ((cpu 0) (interrupts 104952577) (syscalls 428685)
        (pfault 63575) (tlbfault 0) (tlbpurge 0) (load 0) (inidle 99)
        (inintr 0) (systemp 28) (cputemp 22.0) (vtemp 208.0)
        (mbfan 2647) (cpufan 2244) (pwfan 2466)
    )
)
```

The s-expression contains recursively defined s-expressions for each CPU available on the system. For each CPU it recursively defines s-expressions containing atoms carrying the current value of a kernel variable. The values reported in this version of devmon and their data types are summarized in Table 3.1.

This s-expression is completely self-contained and holds all information available for this computer. It can be concatenated with any number of reports from other computer to produce an s-expression describing a set of nodes on a cluster or clusters connected in a grid. With timestamps added in, it can be used to monitor the system status as a time series.

Reading the two other data files currently served by devmon, fans and temp will cause devmon to query the hardware monitoring device on the motherboard and output a single line containing tab-separated integers. For the fans file the three numbers are motherboard fan, CPU fan and power fan respectively, for temp the two numbers are system temperature and CPU core temperature. The motherboard temperature, vtemp reports no value because there is no device connected to it. An example of using fans and temp follows:

```
% cat fans
0       2280    2428
```

Name	Type	Description
sysname	String	The host name of the computer
cpu	Long	The CPU number for which the values are reported
interrupts	Long	The interrupt count since boot up
syscalls	Long	The syscalls executed since boot up
pfault	Long	Page faults since boot up
tlbfault	Long	Translation lookaside buffer faults
tlbpurge	Long	Translation lookaside buffer purges
load	Long	System load; number of processes in **Running** state * 1000
inidle	Integer[0:99]	Percentage of time spent idle
inintr	Integer[0:99]	Percentage of time spent servicing interrupts
systemp	Float	Motherboard ambient temperature
cputemp	Float	CPU core temperature
vtemp	Float	Motherboard temperature
mbfan	Long	Motherboard fan speed, in revolutions per minute
cpufan	Long	CPU fan speed, in revolutions per minute
pwfan	Long	Power supply fan speed, in revolutions per minute

Table 3.1: Data, description and data type format for `mondata`

```
% cat temp
21      28
%
```

The reason for this change in data format are legacy Plan 9 tools, which expect to find tab-separated digits and strings when reading files. Some tools also deduce information from secondary clues, such as the number of lines that a file outputs when read, which is definitely a shortcoming of the current Plan 9 model. In the overhaul of system monitoring planned for the period during preparation of Plan 9 for HPC and clustered environments all those tools will be modified to accept s-expressions [34].

3.2.4 Controlling Devmon

Monctl is used to obtain and control various configuration flags for the device. Reading monctl returns information about settings of mondata output flags. Output flags control mondata's output and decide whether it will write out information about a particular variable it monitors. Here is an example of monctl set to output all possible information:

```
% cat monctl
(
    (context true)
    (interrupts true)
    (syscalls true)
    (pfault true)
    (tlbfault true)
    (tlbpurge true)
    (load true)
    (inidle true)
    (inintr true)
    (systemp true)
    (cputemp true)
    (vtemp true)
    (mbfan true)
    (cpufan true)
    (pwfan true)
)
%
```

Writing to the same file sets those flags via plain-text control messages containing the name of the flag and a keyword, which is either **true** or **false**, as in the following example, ran as the host owner, which queries the number of context switches in the kernel since boot time and turns the display of such information off:

```
# echo 'context true' > monctl
# grep context monctl
    (context true)
# grep context mondata
    (context 14814580)
# echo 'context false' > monctl
# grep context mondata
#
```

3.2.5 Implementation

This section discusses the implementation of devmon's kernel module. Of special importance is to note that the implementation of the entire devmon is extremely small at 343 lines of C code, or 284 physical lines of source code, if empty spaces aren't considered. The usefulness of the 9P protocol as a communication tool is aided by the ease and simplicity of implementing 9P servers, a much easier task for a programmer than a comparable tool for a different operating system. The code for devmon is listed in its entirety in Appendix A.1.

Initialization

Plan 9 does not currently implement loadable kernel modules, therefore devmon is written a kernel device compiled directly into Plan 9 CPU kernels. The `Dev` structure initialized at line 323 contains the name of the device and the Unicode character that can be used to attach to it initially: "∞". It also contains the function definitions corresponding to all device functions, listed in Table 3.2. Whenever a function name does not start with "mon" then the default kernel handler is being called.

The `moninit` function at line 145 is the first function called when the kernel starts up. It initializes the Mon's output flags to print all information and enables the Winbond hardware monitor, discussed in more detail in section 3.3.

Open

`Monattach` at line 318 is called whenever a new client tries to access the device via the `mount` system call. It calls the default `devattach` routine which creates the communication channel for the device and sets its name. In devmon the name is "∞". `Monwalk` at line 158 is a stub that calls the default kernel walk routine since devmon's file structure is static. `Monstat`, `monopen` and `monclose` also use default handling routines.

Read

The most important routine of devmon is `monread` at line 181. `Monread` handles read requests by devmon's clients, grid jobs or just system users. When a client sends a **Tread** 9P request it is handled by the kernel and the `monread` function is called to format the appropriate **Rread** response. The read request comes coupled with a number indicating which file it is attempting to read, held in the `Chan` structure. If the file indicates the root directory of devmon the code generates a list of all files currently served by it by passing the root and the number of elements in the `mondir` structure to the kernel `devdirread` routine. If the file is `monctl` (line 191) devmon will generate a listing of all the possible display flags and their settings. `Mondata` on line 197 prints out statistics from various kernel variables and reads the hardware monitor for temperature and fan speed values. `Qtemp` and `qfans` simply read the sensors on the hardware monitor and print-out their values.

Write

Writing to devmon files is meaningful only in the case when the administrator is writing to the `monctl` file to set or clear any output flags. `Monwrite`, starting at line 273, parses the written data line by line and if it matches any of the preset control messages then the flag is set or cleared according to the command. Writes to all other files return an **Rerror** 9P message.

Close

Closing the device and unmounting it are handled by the kernel.

3.2.6 Data Exchange Example

To illustrate devmon's data path, this section follows data exchange and device operations as they occur during a sample interaction with a user like any of the examples listed above.

In this example a client will read from the `tmp` file using a tool that opens, reads and closes a file, namely `cat`.

1. A client issues a command to access (read) from the `temp` file
2. The kernel's `mount` device, which is responsible for mounting the driver in the first place and handles the initial Tattach message, converts the **read()** syscall into a 9p message of type **Tread** and sends it off through the channel to devmon
3. The kernel which runs devmon (it could be the kernel running the same computer or a different machine connected by a network) transforms the message into a call to the **monread()** routine of the driver, which determines which file the read refers to and formulates a **Rread** response message containing the value of the corresponding variable as its payload or returns an **Rerror** message if access isn't granted
4. The kernel at the originating computer converts the 9P response into raw data completing the system call

3.3 Hardware monitoring

Devmon has been programmed to query data from on-board hardware monitors on motherboards that contain the Winbond W83627THF Low Pin-Count peripheral device [49]. Winbond W83627THF integrates functions such as the disk driver adapter, serial port, parallel port, keyboard controller, hardware monitor and others. The device also provides hardware status monitoring functions which can be used to gather information about several critical system parameters including power supply voltages, fan speeds and temperatures. These functions can be used to support stable and proper operation operation of the computer and to prevent failures. Devmon does not handle power supply voltage monitoring as it is currently meaningless in the context of distributed computing, but in a tightly-coupled cluster environment such information may prove very valuable.

Sensors

The Winbond W83627THF device contains three temperature and three fan-speed sensors. The temperature sensors correspond to CPU core temperature, motherboard chipset temperature and ambient temperature within the case. Of these three the most interesting one is the CPU core temperature, which exhibits the greatest variation (see Figure 3.3). The CPU fan sensors monitor the CPU fan, the power supply fan and a secondary fan that can be plugged in to cool hard drives or simply circulate the air within the computer case. The motherboards available for this research are equipped only with the first two.

The Winbond W83627THF device has the additional ability to raise an interrupt whenever one of its sensors exceeds a high or low watermark. Plan 9 does not implement routines for handling this case, but in the future it may be modified to take action whenever such an interrupt is raised, perhaps by shutting down a computer or sending a message to support staff [34].

Access

Accessing the Winbond W83627THF hardware monitor is accomplished by reading and writing registers in the PCI address space. To read a value from a register we write the register number at the *register index address* and read the value from the *register data address*. Writing a value is done by writing the number of the register it wants to write to in the index address, and the value to the data address.

The register index address for Winbond W83627THF is located at PCI address 0x295, data register is at 0x296. Table 3.3 lists some of the more important registers for the Winbond device. The CPU and ambient temperature sensors are separated into two registers because they report with half a degree of precision.

For example, to enable the hardware monitor devmon writes the value 0x3 to its configuration register, `Cr`. This is accomplished by first selecting the `Cr` register in Register Index, then writing the value to Register Data. Thus devmon first writes the value 0x40 to PCI address 0x295, then the value 0x3 to register 0x296. To read the temperature of the motherboard devmon writes the value of the motherboard temperature register `SysT` (0x27) to the index register `Tidx` (0x295), and reads the data register `Tdata` (0x296) to obtain the reading. Reading and writing via the index and data registers are abstracted in the `wbr` and `wbw` routines at lines 114 and 107 in `devmon.c`.

Visualizing Devmon Data

Figure 3.3 illustrates a visualization of devmon monitoring a Plan 9 CPU server. The graph is created by a version of the `stats` utility which I modified to use information provided by devmon. Stats draws time-based histograms with frequency of 1Hz. The most recent values appear to the right. The graph sections from top to bottom correspond to "system load", "CPU temperature", "motherboard ambient temperature", "CPU fan" and "power supply fan". The drops of the speed of the CPU fan were artificially induced with a sharp, pointy object thrown between the fan blades. The graph illustrates the close correlation between core CPU temperature and the load of the system (or how busy the CPU is).

Adding Other Hardware Monitors to Devmon

Expanding devmon to monitor more hardware devices is quite easy, though limited in my case by the availability of hardware. There are many different vendors of such chipsets, often with conflicting hardware implementations. The `lm-sensors` kernel module [50] provides implementation for many legacy devices and some code from it may be integrated into devmon in the future. Devmon itself has a highly modular design, so adding a new device involves changing a handful of lines of code to recognize the device and call the corresponding read/write routines. If the device is a PCI one devmon can use standard kernel routines to probe and configure it. Examples of such routines used to configure the hardware monitor are found in lines 107 and 114 of the source code listing in `devmon.c`.

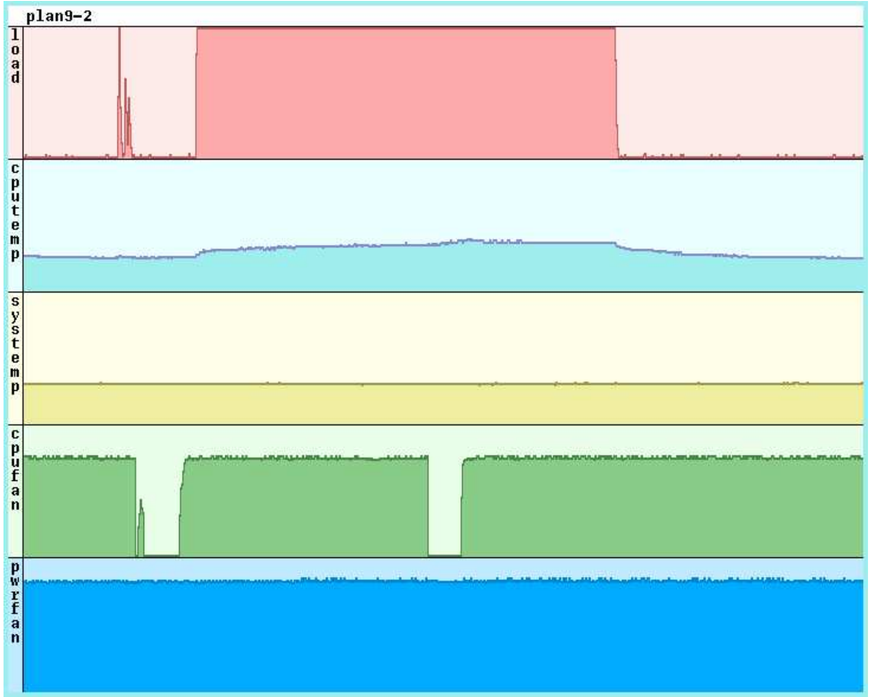

Figure 3.1: Devmon monitoring a Plan 9 CPU server. The first row monitors system load. The second and third rows monitor the CPU and motherboard temperature sensors respectively. The fourth and fifth monitor CPU and power fans. Note how the CPU temperature increases sharply with the system's load, and less sharply when the cpu fan is stopped. The scale on the Y axis is [0,1000], except for the temperature scale, where it is [0,50]

3.4 OS monitoring

The Plan 9 kernel was designed to accommodate operating system monitoring for performance analysis as well as profiling and as such provides valuable performance, execution and statistics counters.

Devmon uses the ones which are deemed to be the most interesting from an evaluation and prediction point of view. I explain them below, with a short description of what extreme values for the particular counter may mean.

interrupts a per-cpu counter for the number of interrupts triggered; a high interrupt count can indicate a faulty hardware device

syscalls the number of syscalls executed by the kernel; a rapidly increasing syscall counter indicates that a large portion of the run time is spent running functions in the kernel instead of running other programs

pfault number of page faults in the virtual memory subsystem; page faults cause system slowdown since the data in the page needs to be fetched from secondary storage

tlbfault the Translation Lookaside Buffer, or *TLB*, is used as a cache for page table addresses; TLB faults mean that memory accesses performed by the program are more expensive

tlbpurge whenever the kernel switches the context of the currently running program with another the TLB must be purged

load system load is calculated based on the number of processes in the **Running** state waiting to be executed; a high number here can indicate that the system is running processes inefficiently or is overloaded with jobs

inidle percentage of the time the CPU spends in idle state; on an optimally performing system this number should approach 0

inintr percentage of the time spent servicing interrupts; any number significantly larger than zero could indicate badly written hardware drivers or a misbehaving device

Devmon accesses those variables via the per-processor struct `Mach`, which describes the system's architecture. For example, to see the number of interrupts serviced by CPU 1 devmon reads `mach[1]->intr`. The complete definition of the Mach structure is given in Appendix A.4.

3.5 Performance of Devmon

Evaluating devmon's performance takes into consideration the fact that there are two basic modes of operation a client can utilize. The first, and by far the most common, is the simple interaction with the files devmon serves via standard Plan 9 utilities and programs such as `cat`, `ls`, `grep`, `awk` and others. It is the one illustrated in previous sections and exemplified by the situation in which a client runs a Plan 9 command such as `cat`, which opens one of devmon's files, reads the information from it, writes it out to a file and exits. The second is when a command opens one of devmon's files for reading, reads it, but instead of closing it seeks back to the start of the file and reads it again, obtaining a new reading from devmon if any of the variables monitored have changed.

The first mode of interaction involves external factors such as loading a binary and executing it, which increases the completion time for the operation. However, since it is by far the most common way of accessing devmon, it has been measured. The measurement was done with a shell script that performed the following operation 1000 times:

```
cat mondata
```

The second mode of interaction gives us a better idea of how well the 9P protocol performs and especially how fast the data path between a kernel and userland is. It was measured by creating a very simple program that performed the following operation:

```
open file mondata;
repeat 1000 times
    read mondata into buffer;
    write out buffer;
    return to start of mondata
close mondata
exit
```

The results, which are summarized in Table 3.2, indicate that the Plan 9 kernel has a very fast data path and that devmon and the 9P protocol are sufficiently fast to saturate a 100-megabit network connection in the case where there are multiple clients. In fact the reason tests with multiple networked clients aren't included here is that the results were

Connection type	Data size (bytes)	Queries/second
cat	208	526
seek	208	25672

Figure 3.2: Maximum number of queries for the two major types of devmon access

purely network-bound. Performing devmon tests on gigabit networks will are planned for the future, pending the completion of gigabit ethernet drivers for Plan 9.

3.6 Devmon and Other System Monitoring Tools

Of the numerous OS and hardware monitoring tools that I have evaluated for my research I found that very few are general enough to work in a grid environment: SNMP is geared towards networks and routers in general and involves the installation of complex implementations; the UNIX /proc is often incompatible between different distributions of different OS vendors and even between different kernel versions of the same OS; programs such as Big Brother [63] are used in web server monitoring and have a monitoring frequency of about half an hour; MRTG [65] is highly configurable but geared towards visualizing trends.

The most useful system monitoring tool for UNIX, sar, part of the SysStat system monitoring suite [66] provides the most information about the current status of the system by interfacing with the `/proc` file system on a host of operating systems. Sar is very similar to devmon in spirit, but there are a few major differences:

- sar relies on its own internal binary data format for storing and retrieving information, thus requiring a sar binary to be present on the host where this information will be accessed
- sar is portable across different UNIX and UNIX-like systems, but it provides a different set of information on each one of them; for example on Linux it reports disk

block read and writes and memory faults, while on IRIX it reports disk block read and writes, memory block reads and writes and cache hits

- sar's binary file formats aren't portable across platforms
- Monitoring several nodes with sar requires special provisions to be made for storing and labeling the data files

3.7 Future Devmon Development

Apart from the features which devmon lacks, most of which concern hardware monitoring (listed in Section 3.3) there is one major area devmon could be extended to: I/O statistics gathering. Distributed computing almost always consists of applications that access the majority of their resources over the network. Knowing which services, networks and files are being used the most could aid in improving 9grid's performance and accounting or give an idea where a job spends the most of its time.

Plan 9 already provides such information on a per-process basis via the `iostats` command, which prints information about processes, but instead of their execution times it lists what files have been read and written during the execution of that process. Plan 9 also keeps information about how much CPU time a process and its children have spent during its execution. Generalizing such information gathering over all processes running on a node will be an interesting, albeit difficult task.

Devmon is the first step in an overhaul of the Plan 9's monitoring tools, planned by me and several other developers [34]. This overhaul will enable faster, better organized and controlled view of the system in clustered environments. The overhaul was proposed by me as an attempt to satisfy the requirements imposed on operating systems by today's heavily clustered HPC installations and the move towards grid computing.

The modifications involve building an integrated hardware and OS monitoring file system for Plan 9. The way it works right now is awkward and difficult to maintain – there are a plethora of files served by different kernel drivers providing information about different parts of the system. Each file usually has its own specific encoding and presents information differently than the others. For example, shown below is the output of two files: /dev/sysstat and /net/ether0/stats. The first gives information about the number of processes and the system load, among others, the second provides statistics about a network interface:

```
% cat /dev/sysstat
      0   400895223   953044579  1223836144     12045184 \
      0           0           0          99            0
% cat /net/ether0/stats
    in: 28451697
    link: 0
    out: 34406060
    crc errs: 0
    overflows: 0
    soft overflows: 0
    framing errs: 0
    buffer errs: 0
    output errs: 0
    prom: 0
    mbps: 100
    addr: 0040058340ba
plan9%
```

As you can see, the formats are different and sometimes incomprehensible. Any program that wants to use such information opens those files and parses the information read from them. This task involves writing several different parsers and leads to some errors, such as the "stats" visualization utility, shown in figure 3.3, not working properly for a few months after the format of the /dev/sysstat file was changed by one of the Plan 9 developers. Nobody noticed the change and the parser was happy to assign 0 to the value it missed.

The change I propose involves taking the relevant bits of information directly from the device drivers and integrating it into a monitoring file system that presents a standardized view of the system's operational statistics. The file system's design is as follows:

```
/dev/stats/
    sysname
    time
    bintime
    memory/
            total
            used
    nusers
    nprocs
    cpu0/
        irqs
        syscalls
        ...
    cpu1/
        irqs
        syscalls
        ...
    ether0/
        in
        out
        err
        ...
    ether1/
        in
        out
        err
        ...
    disk1/
        in
        out
        used
        total
        ...
    disk2/
        ...
```

The directory /dev/stats is arbitrary and simply indicates a possible mount point for the device. Consider /dev/stats/ the root of the file hierarchy. Files in the main directory of devmon provide global information about the computer, such as its name, the current time, the timezone it is a member of, the number of users, the number of processes running, system load and others. Each component, of which there may be multiples on a given computer, has a directory dedicated to it. Such directories are given to processing units, storage disks and devices such as network controllers. Each subdirectory contains files that report information about the single device only, therefore if one wants to gather the total count of interrupts on the system they can sum up the different files in the directory. With s-expressions and a LISP interpreter this can be done via a command formed from the standard shell:

```
; '(+ `{cat cpu*/irqs} )'
```

There are several utilities in existence which accept fully bracketed reverse polish notation arguments and can be used in shell scripts to calculate and graph data. The most important change, however, is that each file reports only a single number to avoid having different formats or having to parse through text.

I have also been working closely with the team developing SuperMon, a lightweight, high-speed cluster monitoring tool, with which devmon is compatible at the data-exchange level. I hope to merge the two architectures and extend them to work in a grid environment in the future.

3.8 Summary

This chapter provided a description of the Plan 9 kernel device *devmon* developed as part of this thesis. Devmon is used in 9grid to present system monitoring information as a network-accessible file hierarchy. The chapter discussed the design and implementation of devmon, the control messages it accepts and the methods for disseminating information that it uses. It also looked at the performance of devmon and how it may fare in a large cluster or grid environment. Devmon was also compared with other system monitoring software solutions.

This chapter also examined how devmon interacts with the hardware monitoring device Winbond W83627THF, which is found on the University of Calgary 9grid computers and used to monitor the cpu fan speeds and temperature sensors on Pentium IV motherboards.

Name	Function
`devreset`	reset the device
`moninit`	initialize the device
`devshutdown`	shutdown
`monattach`	a client is trying to `mount` the file system
`monwalk`	a client is changing its current file to be a file in the file system's hierarchy
`monstat`	a client is trying to find a file's stats information. last changed date created, size, etc.
`monopen`	a client is opening the device for reading
`devcreate`	a client is trying to create a new file in devmon's directory
`monclose`	a file is closed for reading
`monread`	attempt to read from a file
`devbread`	buffered read is attempted from a file
`monwrite`	attempt to write to a file
`devbwrite`	attempt to do a buffered write to a file
`devremove`	attempt to remove a file
`devwstat`	attempt to modify a file's stats information, such as running the `chmod` command

Table 3.2: Functions in the implementation of devmon

Register Name	Register Address	Register Function
Tidx	0x295	Register Index
Tdata	0x296	Register Data
SysT	0x27	Motherboard temperature
Fan1	0x28	Fan 1
Fan2	0x29	Fan 2
Fan3	0x2a	Fan 3
Cr	0x40	Device configuration
Fdr1	0x47	Fan divisor used in calculating rpm
Did	0x49	Device ID
CpuTHi	0x50	CPU temperature sensor bits 8:1
CpuTLo	0x51	CPU temperature sensor bit 0
VTHi	0x50	Ambient temperature sensor bits 8:1
VTLo	0x51	Ambient temperature sensor bit 0

Table 3.3: Important registers for the Winbond W83627THF hardware monitor

Chapter 4

Resource Discovery

Resource discovery in the context of grid computing allow programs to query the existence and features of services available in a distributed environment. On a single-node, non-distributed system resource discovery utilities have existed for a very long time and are generally familiar to users. Programs such as `who`, `uname`, the `proc` file system and others are used frequently to find out the limits and current utilization of various resources. Extending these familiar paradigms to a distributed environment involves design decision addressing various issues such as:

- Security – who is allowed access to the information presented by the resource and operating system monitoring subsystem?
- Data acquisition – what is the best way to gather the data with minimum effect on the system's performance?
- Scalability – how does the system handle an increase in the resources it must monitor?
- Caching – how does the system preserve state between successive connections; what caching mechanisms does it employ to facilitate faster information retrieval?
- Interface – how does the system compare with existing, non-distributed, resource discovery services?

There are two different approaches to implementing resource discovery: creating a brand new set of tools specifically designed to work in a distributed environment, such as the OpenLDAP [33] directory, or extending a familiar set of operating system utilities to work across networks. The standard Plan 9 resource discovery mechanisms take the second approach. OpenLDAP is the current leader in providing resource discovery for legacy operating systems such as UNIX, Linux and Windows. OpenLDAP is open source and is used by the Globus middleware toolkit and most other distributed environments running on legacy systems. It is well understood and provides solid and robust performance that scales reasonably well with the environment. OpenLDAP's role in HPC clusters and supercomputers is relegated mostly to authentication since it is unable to satisfy their very high scalability and performance requirements.

There are several reasons why a port of OpenLDAP to the Plan 9's grid environment, 9grid would not be beneficial:

- OpenLDAP requires a database backend such as Berkeley DB [64], which isn't readily available for Plan 9 and the porting of which would be very expensive for a one-person project

- Some OpenLDAP clients are written in languages unavailable for the Plan 9 platform, such as Java (attempts to port a JVM to Plan 9 have stalled due to the readily available language Limbo, which serves the same purpose)

- OpenLDAP does not fit well with the *"everything is a file"* model employed by all other Plan 9 system resources and programs. To rewrite OpenLDAP so that it presents its resources in the form of files would be just as difficult and complex task as writing a completely new client from scratch

- The overall success of OpenLDAP leaves a large area of solutions to the problem of distributed resource discovery unexplored

This chapter introduces *ResFS*, resource discovery software written as part of this research and used in 9grid. Section 4.1 describes the standard resource discovery mechanisms already available in Plan 9. The design of resfs has been modified to accommodate these mechanisms preserving the feel of the system, while extending its functionality enabling it to work in a grid environment. Section 4.3 provides the reasoning behind the choices I have made in creating resfs, its integration with the rest of the Plan 9 operating system, the requirements it must fulfill, data types and caching policy decisions. Section 4.2 describes the reasoning behind resfs and gives examples of its use. Section 4.4 discusses its features while Section 4.5 talks about the implementation and design of resfs. Performance metrics and analysis are discussed in Section 4.6.

4.1 Standard Resource Discovery in Plan 9

Resource discovery solutions in the context of a single distributed environment have existed since the very first Plan 9 release in 1991. System services exported by the kernel are commonly accessed through a naming convention, depending on whether or not they have been compiled in the kernel. All of Plan 9's devices export a directory hierarchy corresponding to their internal structure and design. Each device has a single-letter name, which is used to refer to it when the first mount is performed. An example of a Plan 9 device driver can be found in Chapter 3 where devmon, a hardware monitoring kernel module for Plan 9, is described.

A list of the currently available drivers that have been compiled in the kernel is generated whenever the file /dev/drivers is read.

```
% cat /dev/drivers
#/ root
#c cons
#P arch
#$ pnp
#e env
#| pipe
#p proc
#M mnt
#s srv
#d dup
#r rtc
#D ssl
#a tls
#B bridge
#E sdp
#K kprof
#l ether
#I ip
#∞ devmon
%
```

The computer in this case is a CPU server so there are a few devices missing, most notably #m and #v (mouse and vga) which are normally used only on terminals. The file directory structure that the device drivers present is bound to /dev during login, therefore in order to discover the basic capabilities of the system a user needs to visit only /dev/drivers and examine this file to find out what is available. It is very common in the course of daily interaction with the system to have a command fail because a kernel resource is not bound to dev or another directory where expected. In such cases a mount command is executed to bind the resource's file system hierarchy from the kernel device into the local namespace. In the cases where the current kernel does not have the device compiled in it could be mounted from a remote system, if that suits the user.

An often used example on Plan 9 systems is to have only one network interface to the outside world and the Internet. When jobs or users want to access external resources they mount the interface over the local /net hierarchy and have all subsequent network traffic routed through there. In the past, when most Plan 9 networks used the IL [8] network protocol and only a few computers connected to the Internet implemented TCP interfaces, importing a remote /net was the only way to access the Internet. The "everything is a file" approach greatly simplifies both the kernel implementation and user interaction with kernel devices and other services. Figure 4.1 illustrates how a user's namespace may be composed of different devices *bound* to predefined places in the system.

User-level servers, which are normally run by user programs and are not compiled in the kernel are discovered in a slightly different manner because they do not have # names associated with them and because communication with them is not initiated through the kernel. Plan 9 allows user-level programs to present their service and resource hierarchy as a named file in a predefined directory. Other programs wishing to use those services rendezvous at this directory and mount the directory structure rooted at the named file in their own namespace.

A special kernel device called srv, which serves as a filter relaying messages between two namespaces, facilitates this rendezvous. A process willing to share its namespace with others can *post* a file descriptor pointing to the root of a namespace via the srv utility to a specialized location on the system, namely /srv. Other processes then mount the

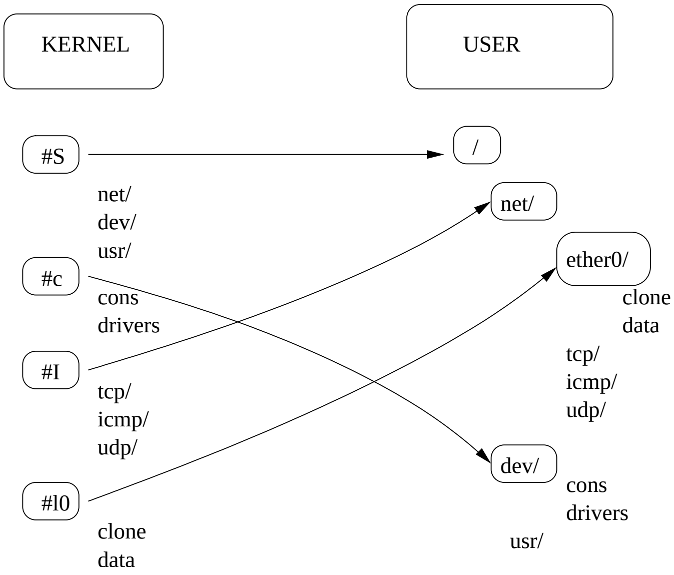

Figure 4.1: Interaction between kernel device drivers and a user's namespace. Arrows correspond to `mount` calls issued by user programs.

desired file server hierarchy from `/srv` if file permissions allow.

The following is an example of `/srv` on a Plan 9 terminal serving a single user named `andrey`. The files correspond to directory structures served by the the window system (`rio`, `riowctl`), the security agent (`factotum`), the plumbing mechanism used for context-dependent program execution, the text editor and the domain name resolver:

```
% ls /srv
/srv/acme.andrey.112
/srv/boot
/srv/cs
/srv/dns
/srv/factotum
/srv/plumb.andrey.78
/srv/rio.andrey.16
/srv/riowctl.andrey.16
/srv/slashn
%
```

At any point, provided that file permissions allow it, a user can mount the directory pointed to by one of those files, or serve a part of the user's namespace as a named file:

```
% ls /mnt
% mount /srv/factotum /mnt
% ls /mnt
/mnt/factotum
% ls /mnt/factotum
/mnt/factotum/confirm
/mnt/factotum/ctl
/mnt/factotum/log
/mnt/factotum/needkey
/mnt/factotum/proto
/mnt/factotum/rpc
%
```

Figure 4.2 illustrates how two users can share a namespace through the "bulletin board for file descriptors" – /srv; both users mount the #s device driver, user A *exports* the /exp directory as a.srv, user B mounts a.srv in their own namespace.

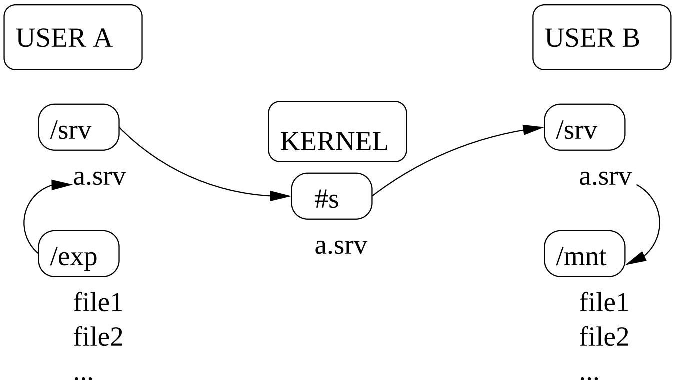

Figure 4.2: Two users sharing a namespace through /srv. Both users mount the #s device driver. User A *exports* the /exp directory as a.srv. User B mounts a.srv in their own namespace.

In Plan 9 virtually everything is treated as a file in a directory hierarchy, therefore it is easy to extend the /srv paradigm from the single system domain and have it work across the network. In fact in Plan 9 the source code maintainers use exactly this scheme to present system updates to Plan 9 users around the world. A person who wishes to update their system installation imports the developer-maintained tree from sources.cs.bell-labs.com and presents it as a subdirectory in the current namespace. Finding out whether a file has been updated is as simple as comparing timestamps or running a diff between the two directories.

Plan 9 also provides a *Network DataBase* service, abbreviated NDB. NDB serves as the main name resolver when establishing a connection between various services on the system installation and the computers providing those services. A typical usage of NDB is to query for various commonly used components of a Plan 9 installation such as authentication, CPU and domain name servers. NDB is aware of networked domains and memberships, which makes it very useful for describing hierarchical structures such as 9grid. The following is an example of a typical real-life NDB configuration as used by the University of Calgary Plan 9 installation:

```
ipnet=hidden ip=192.168.0.0 ipmask=255.255.0.0
    proto=tcp
    cpu=plan9.ucalgary.ca
    fs=plan9.ucalgary.ca
    auth=plan9.ucalgary.ca
    authdom=plan9-hidden.ucalgary.ca
    resourcefs=plan9
    dns=136.159.5.14
    dns=136.159.5.15
    dns=128.233.3.1
    dns=128.233.3.2

sys=plan9-2 ip=192.168.1.3 ether=0040f46ed68f
```

The first entry describes a network behind one of the University of Calgary firewalls. The network assumes a local IP address. The proto entry specifies that terminals and CPU servers should connect using the TCP protocol instead of the Il one. Cpu, fs, auth and dns specify the addresses of computers which provide those services. Authdom is the authentication domain to which this network belongs. The last line is a sample entry for one of the computers belonging to this domain.

An NDB dæmon is started by every Plan 9 user as they log in to the system. The NDB binary provides a control and data file named /net/ndb which the user can read to query information. Writing to the `/net/ndb` pairs of the form *name=value* will result in *value* being returned whenever *name* is queried. Outside of general Plan 9 administration NDB is used by resfs to cache information about remote grid nodes, as will be explained in Section 4.3.

There are a couple of issues with the current state of resource discovery in Plan 9 that prevent it from being extended across grid environments. One is the lack of centralization in reporting the system resources available, which I have addressed in Chapter 3 with `devmon`. The other is the lack of integration between system monitoring (devmon), the network database services providing query services for resources (ndb) and the networked environment spanning multiples of computers from different Plan 9 installations. *ResFS* is created with the task of binding those components together and providing easy access to information about resources on the grid.

4.2 ResFS

ResFS is a hierarchical file system utilizing the 9P protocol to present a set of directories corresponding to computers on a network and files containing information about resources on those computers. Resfs is a resource discovery mechanism for network-connected environments such as 9grid.

The architecture of resfs has two components, namely *leaf* and *aggregate* nodes. The leaf nodes usually reside on computation servers and announce themselves on one or more registry services, which act as aggregates of information about a subset or a complete set of nodes available to an administrative domain. Aggregates themselves can be mounted together hierarchically as administrative groups combine to work together. Registry services can also report to other 9grid registry services creating a global view of the grid's available resources.

Figure 4.3 illustrates one possible combination of grid resources. *A*, *B* and *X* are administrative domains. *A'* and *B'* are alternative registries, local to their respective domains. Nodes run resfs directly, which creates a file system containing files reporting the architecture of the node, the number of CPU's available and the number of users on the system at the current time. *GridCtrl* is a service that could be exported to the outside world containing information about the entire infrastructure of the grid domain. Users *mount GridCtrl* to gain access to the files exported by nodes on the entire grid provided they have permissions to do so. There is no restriction on the number of grid controllers one can have, i.e., GridCtrl does not hold exclusive privileges over the data and users are allowed to query local registry services, such as A or A' whenever they need access to a subset of the information, or one that external users have no access to. Users can also query a computational node by accessing its resfs directly, though in this case they need to know the node's name or its address on the grid.

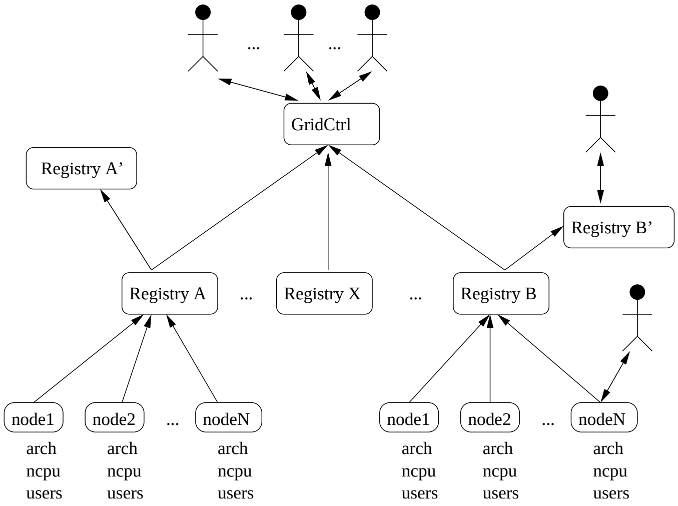

Figure 4.3: A sample resfs structure in 9grid. A, B and X are administrative domains. A' and B' are alternative registries, local to their respective domains. *GridCtrl* is the main grid controller.

Resfs presents a very flexible infrastructure without a superficially imposed order and hierarchy. Another possible arrangement of resfs registries and nodes is illustrated by A' and B', which can be thought of as registries offering an alternative, restricted view of the systems for users who aren't allowed access to the full view available from GridCtrl.

The arrangement of nodes and aggregates in resfs can match a wide variety of needs and configurations. The only real limits are network latencies. For about four months in 2004 the Plan 9 installation maintained as part of the 9grid project at the University of Calgary was used as a resource discovery station for computers around the world. A single resfs aggregate registered clients from all over the world, including Finland, USA, Bulgaria and Russia. Of course a single query for all computers in this environment took longer to complete due to the latency of overseas networks and the fact that some computers were on very slow connections. The latency of slow overseas connections can be overcome by caching most of the data in the local per-user NDB database, or globally, per domain, by the domain administrator. The NDB database is detailed in Section 4.1. The caching policy decisions for resfs are described in Section 4.3.

4.3 Requirements

Resfs is a step towards creating administration tools for a large set of widely distributed resources, unifying them in a single environment. This environment is similar, in terms of features, interface and security, to the private namespaces which unify the resources available to a single Plan 9 installation on a per-process basis.

The justification for creating resfs stems from the fact that even though Plan 9 is a distributed operating system, meaning that it has the ability to connect many remote resources into a cohesive environment, it was never designed to handle clustered and cross-administrative environments. Even though it compartmentalizes function into separate networked hardware it still aims to provide a single system image across all the computers connected to it. A typical Plan 9 installation therefore is comprised of many separate nodes, but presents each one with the same view of the system. This is a consequence of Plan 9 being designed in a time when it was expected that the number of processors available in a single, mainframe-like computational unit would far exceed the number of users of that system [32]. In grids, on the other hand, users prefer to preserve the localized view of their workstations, which in most cases are highly personalized, and utilize remote resources only when the need arises.

Plan 9 provides a few benefits other systems currently do not have, most notably the simplicity and elegance of the 9P protocol, the protection of private namespaces in multi-user servers and workstations and the seamless integration of resources on the local network into the environment. The file server interface where everything is a file reduces the complexity of resource discovery on the grid to the simple task of examining the contents of files stored in a hierarchical directory tree, something most users are intuitively familiar with. This interface also alleviates the burden of having to create multiple new clients to allow access to the data. Writing specialized tools to make the data provided by resource discovery mechanisms more presentable is a task that each grid administrator has to undertake at one point or another. Java clients or perl scripts are commonly used to extract the useful information from databases. A Plan 9 user familiar with the shell, sed and awk is able to create much more powerful scripts adapted to the particular job at hand very quickly. Having powerful shell one-liner commands at one's disposal gives a familiar "shell" feel, facilitated by its programmability. Users wishing for a more user-friendly interface are welcome to use any file and directory browser available, including a web browser able to navigate through a file hierarchy.

Programmability is key in this system, especially in aiding programming using alternative languages. There are no extra bindings required for working with resfs' data, however there are several types of languages which benefit from its data representation more than others:

Regex-based Languages such as perl, awk, tcl, sed and others, which rely on string manipulation using regular expressions benefit from the plain text representation of all data in resfs.

LISP *S-expressions* [38] are easily manipulated in LISP. This was done in order to more easily integrate resfs with SuperMon [36], the High Performance Cluster Monitoring Tool.

NDB Text output is formatted to be acceptable as a system node definition by Plan 9's Network DataBase service, described in Section 4.1, which is the standard tool used to query network and environment information about the system.

On the technical front, resfs benefits by utilizing a scheme which does not force nodes accumulating information about computers on the system to poll for information from leaf nodes. Neither does it require leaf nodes to push such information out to the aggregates. There is no midway storage anywhere on the system. There is no knowledge exchange of the type and quantity of data passing between clients and leaf nodes. This means that, besides the initial handshake between a leaf and an aggregate. There is no information being exchanged between the two, except in the case where a leaf node goes offline. Resfs aggregates therefore serve as a vector containing pointers to other systems' local resfs nodes. Its knowledge base is limited to knowing which computers are currently alive on the network.

A step further into the realm of grid computing is the "no configuration required" design of resfs' aggregate nodes. This means that as long as the aggregate is functional there is no limit to the number of CPU nodes that can be functional at any time, except the limits imposed by the 9P protocol. There is no configuration step for any nodes in resfs in fact; as soon as an aggregate is operational and the system is configured with its location any number of leaf nodes can register to it. To visualize this one can think of aggregates serving as bidirectional pipes between the client and the resfs running on a leaf node on the system. The main purpose of the aggregate is to provide an easy way of looking up the nodes available in the domain. It will not try to guess the user's desired actions by taking part in controlling leaf nodes or caching information about them. The user has full control of how much of the information available should be queried and retained via NDB.

This simple design leads to some surprising performance results such as the resilience to overload of aggregate nodes. When attempting to overload the system with clients the computers where the clients resided were overloaded first. Naturally a more complex system is desired, where an aggregate node contains information about the sum of all nodes reported to it. For example an aggregate node for an administrative domain could report information about the number of *all* available CPUs from all leaf nodes that have reported to it. It could also report all the different types of architectures that it has access to. This can very easily be achieved via an s-expression parser built into the aggregate node, however it was omitted due to the relatively slow connection and wide geographical distribution of organizations in 9grid, with most of the users being individual single-node Plan 9 installations. Trying to provide up-to-date information for leaf nodes separated as far away as Japan and Finland would slow down the aggregates unacceptably.

Resfs aggregate and leaf nodes avoid caching on purpose, aiming to provide only the most up-to-date information available. This behavior is correct in the cases where making decisions based on outdated information can prove costly. An example of such a scenario is the case where a computationally expensive job is started on a set of nodes based on an outdated load average reading immediately after another job has started there. The two jobs are now competing for resources causing a delay in their completion.

Looking from the other side, there are many types of resource information that need not be queried frequently on account of them being static most of the time. Such resources include a host's name, IP address, operating system type or architecture. A client would not benefit by having to access such information often, even less so if the data happens to be accessible only via a slow, high-latency connection. Resfs solves this problem by utilizing Plan 9's network database, NDB. The information provided by resfs leaf nodes is given in a format accepted by NDB as a single node's setting of *name=value* pairs. Simply redirecting the information output by resfs' `stats` file onto the local `/net/ndb` file will cause this information to be cached locally. Updating the local copy of the remote node's resfs is as simple as performing the same operation twice, or using any of the available NDB commands to set a specific *name=value* pair only.

4.4 Features

Resfs has a multi-tiered architecture where parts of the system assume different responsibilities depending on the task they have in resource discovery. There are two essential types of roles, reporting and aggregating. Compute nodes, which typically play the reporting role, deliver status information when queried. Aggregates contain meta-information about the location and availability of compute servers without holding or taking part in the communication of data between client programs and leaf nodes. By mounting the file systems exported by leaf nodes onto aggregates one can build hierarchies of aggregate nodes. An entire grid can then be queried in a simple manner by recursively traversing the directory structure from the top level.

4.4.1 Leaf Nodes

A typical leaf node reports information about the status of the system it is running on. It generally includes information such as the number of processors, the hardware architecture, amount of memory available and system load. This information is presented via a file system interface which can be accessed using one or more of the following methods:

mounted locally by a user process running on the same system

imported remotely via a process running in the same administrative domain as the system

imported remotely via the aggregate node responsible for the resfs system on this administrative domain

The following demonstrates a sample session with resfs running on the local computer. The user examines the `/srv` directory to see if there is a resfs instance running. The running instance has announced itself under the name `resfs` and the user mounts it under `/mnt/resfs`, which is previously shown to be empty:

```
home% ls /srv
/srv/acme.andrey.112
/srv/boot
/srv/cs
/srv/dns
/srv/factotum
/srv/plumb.andrey.78
/srv/resfs
/srv/rio.andrey.16
/srv/riowctl.andrey.16
/srv/slashn
home% ls /mnt/resfs
home% mount /srv/resfs /mnt/resfs
home%
```

At this point the user has made a connection to the resfs instance and has obtained a file descriptor/channel pointing to the root of its file hierarchy. The user runs a simple command that lists the files available for queries. Note the file permissions and user ownership. Resfs is being run by a single user, `andrey`, and does not allow any information to be written to any of its files. Most files have familiar names indicating the information they report.

```
home% ls -l /mnt/resfs
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/ctl
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/load
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/memory
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/ncpu
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/nproc
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/nusers
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/objtype
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/os
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/s-expr
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/stats
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/swap
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/sysname
--r--r--r-- M 115 andrey sys 0 Feb  2 17:18 /mnt/resfs/uptime
home%
```

Examining the files using `cat` yields data about the current status of the system:

```
home% cat /mnt/resfs/load
load=0
home% cat /mnt/resfs/memory
memused=6942 memtotal=24268
home%
```

A simple script illustrates that the data is repeatedly updated and can be continuously queried:

```
home% for (i in 1 2 3 4 5 6) { cat /mnt/resfs/uptime; sleep 1 }
up 2 days 08:57:58
up 2 days 08:57:59
up 2 days 08:58:00
up 2 days 08:58:01
up 2 days 08:58:02
up 2 days 08:58:03
home%
```

It can also be queried without closing the file by calling `seek(0)` and re-reading the file again. There are no clients that currently implement this method except for test purpouses.

The information presented in resfs' files is explained below.

ctl the only writable file in resfs. When read the `ctl` describes the entire system,including any policy that may have been set in a format usable by Plan 9's network database, *NDB*. When written to, it accepts `value=variable` pairs to be set in the local *NDB*.

load when read, this file gives a numerical representation of the system load, which is arrived at by using the formula $1000 * p * n$, where p is the number of processes in the running state for the time interval, and n are the number of processors available on the system

memory when read gives the currently used and total amount of memory available on the system in blocks of 4096 bytes

ncpu the number of CPU's available on this system

nproc the number of processes in the system, control over those processes is achieved separately by importing the `/proc` file system

nusers the number of users logged in to the system

objtype the architecture of this particular installation, one of *i386, alpha, powerpc, mips, arm* and possibly others

os the operating system type

s-expr same as `ctl` but given in *s-expressions* [38], which would allow the data to be used in a cluster environment with *SuperMon*

swap the used and total amount of swap space available on the system, in block sizes identical to the ones reported by `memory`

sysname the name of the computer

uptime the amount of time since last reboot for the system

The file interface is static, but it is possible to extend it in the future by dynamically creating files to be monitored for data.

4.4.2 Aggregate Nodes

Aggregate nodes do not have a structure or a file interface besides a control file `ctl` which accepts a few simple commands which manage the connected leaf nodes, cause the aggregate node to report to a main controller node or to dump information about leaf nodes into the *NDB* database.

The other file served by an aggregate node is a directory used to mount all leaf nodes which have reported that they are operational. A typical resfs directory on a small grid of three nodes, one aggregate and two leafs, with the aggregate node also running as a leaf node, is shown in Figure 4.4.

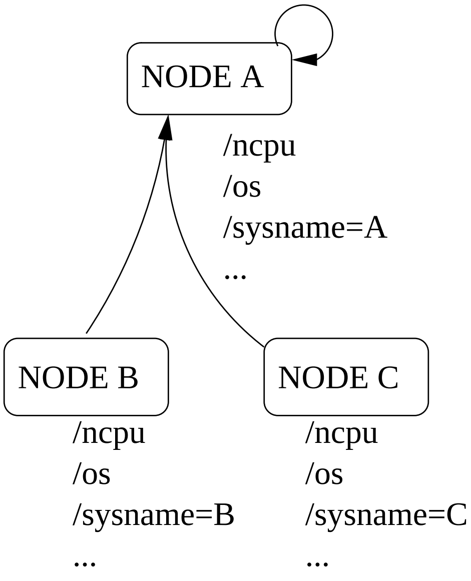

Figure 4.4: An example of resfs running on a small system of three nodes. *B* and *C* are leaf nodes who have registered with *A*. *A* is also running its own leaf node server.

The directory structure visible to users who have mounted the aggregate in this case will be:

```
home% ls /mnt/resfs
/mnt/resfs/ctl
/mnt/resfs/A
/mnt/resfs/B
/mnt/resfs/C
home% ls /mnt/resfs/*
/mnt/resfs/A/ctl
/mnt/resfs/A/load
/mnt/resfs/A/memory
/mnt/resfs/A/ncpu
/mnt/resfs/A/nproc
/mnt/resfs/A/nusers
/mnt/resfs/A/objtype
/mnt/resfs/A/os
/mnt/resfs/A/s-expr
/mnt/resfs/A/stats
/mnt/resfs/A/swap
/mnt/resfs/A/sysname
```

```
/mnt/resfs/A/uptime
/mnt/resfs/B/ctl
/mnt/resfs/B/load
/mnt/resfs/B/memory
/mnt/resfs/B/ncpu
/mnt/resfs/B/nproc
/mnt/resfs/B/nusers
/mnt/resfs/B/objtype
/mnt/resfs/B/os
/mnt/resfs/B/s-expr
/mnt/resfs/B/stats
/mnt/resfs/B/swap
/mnt/resfs/B/sysname
/mnt/resfs/B/uptime
/mnt/resfs/C/ctl
/mnt/resfs/C/load
/mnt/resfs/C/memory
/mnt/resfs/C/ncpu
/mnt/resfs/C/nproc
/mnt/resfs/C/nusers
/mnt/resfs/C/objtype
/mnt/resfs/C/os
/mnt/resfs/C/s-expr
/mnt/resfs/C/stats
/mnt/resfs/C/swap
/mnt/resfs/C/sysname
/mnt/resfs/C/uptime
home% cat /mnt/resfs/*/sysname
sysname=A
sysname=B
sysname=C
home%
```

4.5 Design

This section discusses the design considerations for resfs.

4.5.1 Leaf nodes

Resfs is about a thousand lines of source code written in the C language using libraries common to the Plan 9 operating system, but which are also available for legacy systems such as UNIX and Linux made available via the Plan9Port project [58]. The files are given in appendix A.2 and correspond to:

- `resfs-th.h` – header file
- `common.c` – common operations
- `policy.c` – policy support functions
- `resfs-th.c` – main threaded code
- `stats-th.c` – statistics and monitoring

A leaf node in resfs is designed as a threaded application which could either be executed from the `cpurc` boot script, or started by hand from a user connected to the system. Using `cpurc` is the preferred case, since it will report itself to the local aggregate node immediately after a reboot requiring no human intervention.

The static 9P file hierarchy is built at runtime by the binary and handed off to the 9P server thread, which manages messages received by clients such as `attach`, `walk`, `open`, `read` and `close`. It also takes care of maintaining a set of unique `Fid`'s for each client's request, as per the 9P protocol requirements [40].

All network connection requests are handled by the network thread, which refers the network file id to the 9P server thread, but is otherwise not involved in the 9p communication.

Initialization

Resfs starts in the entry point of `main()` at line 276 of `resfs-th.c` where it parses its arguments (lines 283-307). The arguments accepted by resfs are:

-D print 9P protocol transactions to standard output

-d turn debugging on

-s name announce under name *name* in `/srv`

-P disable announcing to aggregate nodes

-u disable the update thread; information given by resfs will not be updated

-q do not listen on the network

-t num sleep time for update thread; *num* is in microseconds

After parsing the arguments resfs allocates a `Tree` structure which describes the file system served by it (line 321) and creates the root file node, a directory pointing to the beginning of the hierarchy (line 323). Next it populates the `Tree` structure with the file names of the files it serves (lines 328-333) and if not disabled will call `listensrv()` and `calagg()` which establish a listening process for serving users' requests and call the aggregate node to announce that the leaf node is up. The aggregate node's system name is obtained via the local NDB, which should be configured beforehand by the node administrator. For 9grid it is normal to add the node which will be running as an aggregate as part of the network configuration for the remote domain. The NDB configuration for 9grid domains usually includes the system names and IP addresses for the respective CPU and auth servers, as well as the authentication domain and resfs system name. Having completed the initialization stage of its life, resfs calls the `update()` function (line 342) which establishes the update thread and then exits.

The Update Thread

The update thread (line 133 in `stats-th.c`, Appendix A.2.5) handles gathering the system monitoring information from different sources on the node. First it forks the update thread (line 140), then reads values from various files in the system and fills up the Machine structure for this computer. After it has done so the thread sleeps for a predefined time and initializes the information again in a loop.

Client Requests

Client requests to resfs are generally handled by the network thread in the function `listensrv()`, line 229 of `resfs-th.c`, appendix A.2.4. `Listensrv()` starts off by forking itself from the main program (line 235) and establishing a listener to port 18000 (line 244), then it loops waiting for remote connections. When a client arrives `listensrv` forks a process that will handle responding to the 9P messages received from this particular client (line 258). The network call is accepted at line 262 and the standard library routine `srv()` is called, which will handle calling the appropriate routines whenever `read`, `write` or `walk` requests are sent by the client; those routines are listed in Appendix A.2.2.

4.5.2 Functionality

There is no constraint on the information that resfs could access when started by a user who can log in to the computer, since the default Plan 9 installation is fairly permissive when it comes to reporting resources. In fact, all information available through a running resfs instance is available to all users who can log in to the particular computer; resfs only accumulates it and presents a unified interface for all computers on 9grid.

Resfs has the ability to update the data it reports at different intervals, the default being 1Hz, i.e., once a second; an argument, `-t`, provided at runtime by the user who starts resfs changes that value. The smallest update quantum in the current implementation is 1 microsecond. This results in 1000Hz updates, which is approximately twice the amount of time it takes to process two consecutive 9P read requests, assuming they are independent and need to perform a `walk` through the resfs hierarchy.

The leaf node utilizes the very lightweight and robust threads implementation in Plan 9, a descendant of the threaded language Alef [39]. Three threads are running at each time, the 9P thread handles 9P requests, the update thread writes to a shared structure corresponding to the computer representation that resfs maintains and the network thread manages client requests on the network port resfs listens to. Figure 4.5 illustrates the organization of the leaf node.

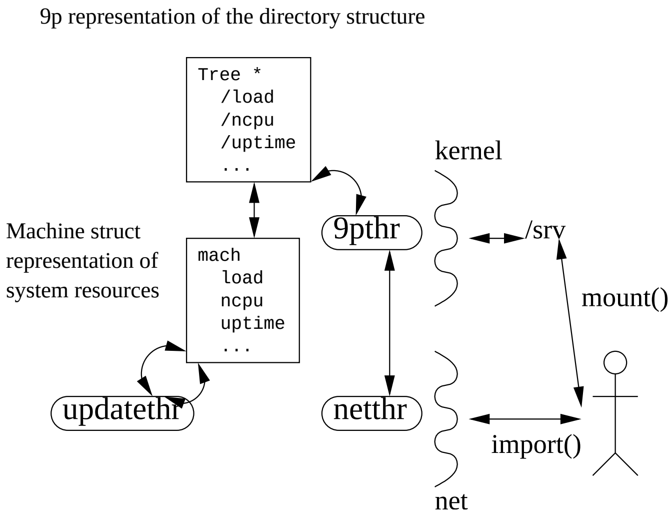

Figure 4.5: resfs thread representation: `updatethr` handles the update of the computer representation data structure, `9pthr` reads and responds to 9P messages using a `Tree *` structure, netthr listens on port 18000 and accepts requests from network clients.

The update thread is necessary in order to account for the fact that querying system resources from outside of the kernel is more computationally expensive than serving the query from local memory. If the update thread is removed a single request would take much more than 0.003 seconds to complete, since that involves at least four extra system calls: opening a file, seeking, reading and closing it. Updating "off-line" with respect to the query means that the data is stored in easily extractable form and the program does not parse the same data over and over again for different requests, which could cause a denial of service. Having the `update` thread also frees up resources in case the leaf node is flooded with requests. Most of the files resfs accesses involve kernel data structures requests through kernel device drivers and avoiding frequent use of those frees up computational resources.

A source code listing for our prototype resfs leaf node is given in Appendix A, Section A.2.

4.5.3 Aggregate Nodes

Aggregate nodes have a design very similar to the leaf nodes. The exception is that they serve only a single control file themselves. Aggregates provide glue between the leaf nodes and the clients. They could be thought of as a collection of union mounted remote directories, each pointing to a leaf node resfs instance running on a remote computer connected to 9grid.

Aggregate nodes start up the same way as leaf nodes, with the exception that they do not have an update thread. Aggregates do not use static `Tree *` structures to maintain the directory entries, instead they use a dynamically generated set of `Fids` for each leaf node server that registers with the aggregate.

Whenever a connection to the leaf node is dropped, the aggregate uses a feature of the 9P protocol to remove the file whose connection is timed out sending the so-called `clunk` message back to the client. This ensures that even when computers and networks may be going offline at any time the worst that clients will see is a very small delay in the response, normally less than a second.

The aggregates handle commands written to the `ctl` file. The commands currently available are listed below:

dump [node] dump the information from node **node** into the `NDB` repository for the system

set variable=value set variable=value in the local ndb database

clear [node] delete a node connection, or all nodes; this is equivalent to running `rm node` in the directory of the aggregate

quit quit the system, closing all client connections

The `ctl` file is write-protected by anyone other than the person who started the aggregate, which is usually the host owner of the computer. The source source code for the resfs aggregates is listed in its entirety in Appendix A.3. Note that at 165 lines of code it is extremely simple.

4.6 Performance

The performance of the early resfs prototype proved to be significantly better than expected. It is matching, if not exceeding, the performance of similar tools such as OpenLDAP [33] and SuperMon [36], this being a testament to the benefits of having simple protocols handling the underlying data connections.

Tests indicate that in its very basic state, with a single lock per computer structure or system description and a leaf node data update per microsecond resfs is able to serve in excess of 600 basic queries per second, or at a rate of 600Hz. When the sampling rates are increased from 1Hz gradually to 1000Hz the querying speed decreases linearly. Figure 4.6 plots the queries per second versus the interval between different updates. It shows that there is a fine line between how much information can be obtained for a given interval and how often this information can be updated within resfs itself. Observing the same patterns when querying resfs directly via the `seek()` system call (Figure 4.6) indicates that the system is sensitive to the number of updates performed per given time interval. An update interval smaller than 100 microseconds will cause the values reported by the leaf node to measure resfs itself predominantly, superseding any other jobs running currently on the system. A solution to this problem can be achieved by imposing hard restrictions on the minimum time between resfs updates from sources outside of resfs' control, such as external drivers or protocol interfaces. A basic optimization similar to this is already built in resfs and it will not query persistent information such as the IP address of the node or the system's name more than once. The test set-up for Figure 4.6 involved wallclock timing of 1000 queries achieved by executing the command `cat load`.

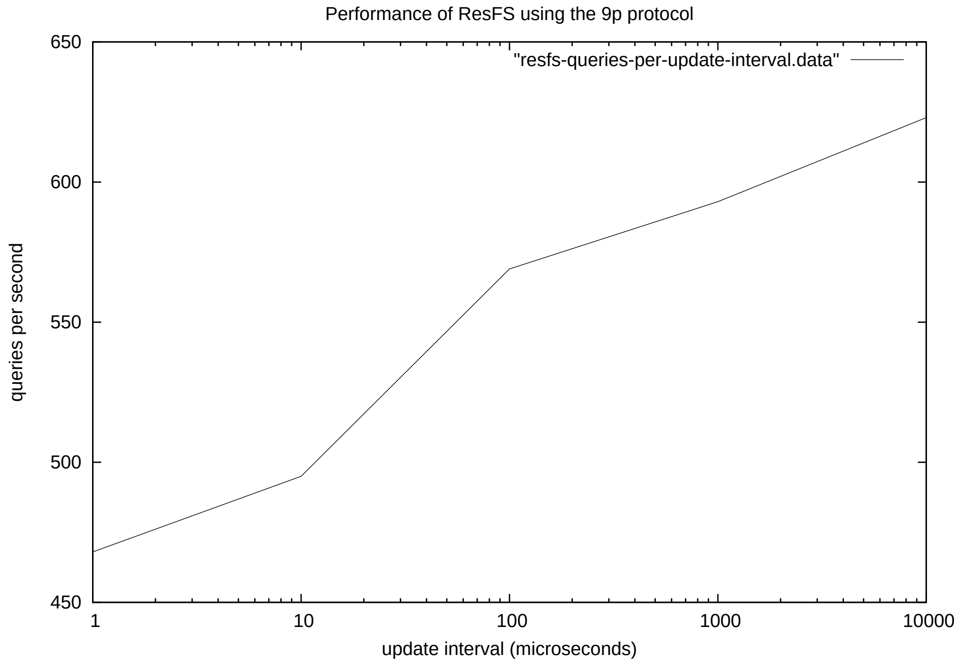

Figure 4.6: Maximum number of queries per second performed to a local resfs server, versus update interval for the update thread in microseconds.

```
#!/bin/rc

for (i in ‘{ seq 1 1000 }) { cat /mnt/resfs/load }
```

Figure 4.7: Testing resfs from userland

Resfs is mounted locally, the cat binary given in Figure 4.7 is also local, i.e., there is no file server involved. The script in Figure 4.7 is the minimum useful program written in a shell scripting language. The reason that the command is executed 1000 times is to avoid measuring the time needed to parse the script.

The test set-up for Figure 4.8 involved wallclock timing of 1000 queries achieved. The test program is listed in Figure 4.9. The program works by opening a file from a hierarchy served by resfs which was previously mounted at `/mnt/resfs`. The file is then read in a buffer and immediately written out to the program's standard output channel. The program then immediately seeks back to the beginning of the file and reads it again,

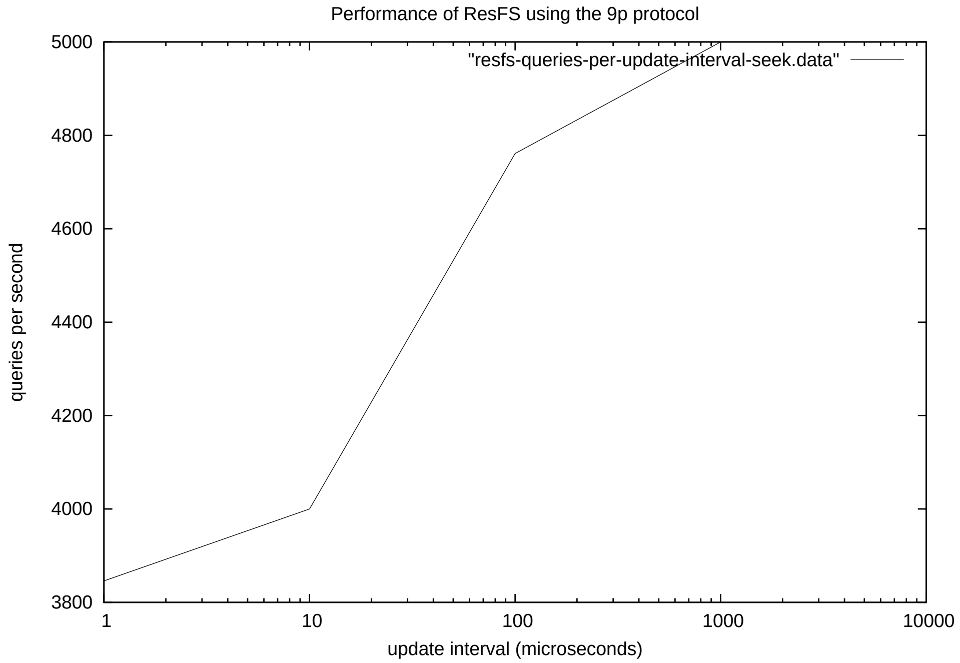

Figure 4.8: Maximum number of queries per second performed to a local resfs server, versus update interval for the update thread in microseconds using seek().

obtaining new values. This is the absolute minimum code one needs for a useful resfs reader without resorting to writing it in assembly language.

For pure CPU servers, where the `cat` binary is fetched from a remote computer, the timing was around 34 seconds with very little deviation, this means resfs was able to serve 292 requests per second. The bottleneck in this case was the network connection, which became saturated from fetching the 20K of the `cat` binary being read from the file server over and over. It made virtually no difference what sleeptime was for the update thread.

Performance data is summarized in Table 4.1. The difference between a single and multiple walks was the amount of 'walk-ing' that needed to be done to access the file. Each walk message is a 9p request, and in the case of `/mnt/resfs/load` there are 6 of them, with 'load' there is only one.

On the other hand, it was possible to simulate in excess of a thousand nodes registering per aggregate without a significant decrease in speed. The 9P protocol has been tested

```
#include <u.h>
#include <libc.h>

void
main(int, char *)
{
    int i, fd, n;
    char buf[1024];

    fd = open("/mnt/resfs/plan9/sexpr", OREAD);
    if(fd < 0)
        sysfatal("can not open resfs file: %r");

    for(i = 0; i < 1000; i++) {
        while((n = read(fd, buf, 1024)) > 0)
            write(1, buf, n);
        seek(fd, 0, 0);
    }
}
```

Figure 4.9: Test program for evaluating synthetic file system performance

FS type	Connection type	Update interval	Walltime (10000q)	Q/s
fossil	single walk	1μs	19.0s	526 q/s
fossil	multiple walks	1μs	20.6s	485 q/s
ramfs	single walk	1μs	17.7s	565 q/s
ramfs	multiple walks	1μs	19.4s	515 q/s

Table 4.1: Queries per second for various types of resource access.

successfully on thousands of nodes [34] and shouldn't be a significant obstruction to scalability. Fine-grained locking, directory traversal optimizations and faster data-to-file and file-to-data conversion routines should also aid in performance improvements.

In the future resfs will move to the lock-less Plan 9 threading library [9] which I discovered too late to to be able to successfully convert resfs to it in sufficient time in order to include it here. The new model simplifies programming even further enabling threads to synchronize via messages sent on channels.

4.7 Summary

This chapter introduced *resfs*, a resource discovery system created for and used in 9grid. This chapter discussed the tools already existing in Plan 9 to provide similar features in the case of a single installation and how they are expanded and integrated with resfs into a tool useful for the vastly distributed resources utilized in grid computing environments. Resfs is created with simplicity in mind and this chapter justifies the choices made with regards to the data types used, connection methods and caching policies. It also discusses the main features built into resfs and how they can be used to program the output data it provides into a personalized view of the grid's resources.

The chapter also described the implementation of resfs in the C language using programming tools available on the Plan 9 operating system. It also provided performance measurements and evaluation of resfs.

Chapter 5

Summary and Future Work

5.1 Summary

This thesis described a set of tools created to aid the building and management of grid computing environments based on the Plan 9 from Bell Labs operating system. They provide features required to support large scale distributed and cross-administrative domain grid computing.

Chapter 1 introduced this thesis and the topics of grid computing it covers. It defined the terminology used throughout the thesis and discussed the major motivation behind the work, which is to facilitate the creation of simple and easy to use environments connecting together physically separated computers and enabling resource sharing between individuals and organizations. Chapter 1 also gave a historical overview of the developments in the field of distributed computing which lead to the creation of the grid computing paradigm. On the software side the chapter examined other solutions that allow resource sharing on a large scale and compared the goals of such projects with the goal of 9grid, the distributed environment build around the Plan 9 operating system. The chapter also outlined the list of contributions this thesis strives to accomplish.

Chapter 2 introduced the *Plan 9 from Bell Labs* operating system and the distributed computing features it provides. It gave a historical perspective of the OS and how design decisions affected features such as security, authentication or remote service access. It introduced the concept of private namespaces and gave examples of how private namespaces help in securing the Plan 9 environment. The chapter also discussed the single sign-on authentication agent *factotum* and its role in providing seamless authentication in a distributed environment. The chapter also described 9P, the communication protocol on which all Plan 9 services are based, both local and remote and discussed some additions to the Plan 9 operating system to help its adoption in grid computing, such as a new authentication mechanism that crosses administrative domains and a groupware communication service, called chatfs.

Chapter 3 gave a description of the Plan 9 kernel device *devmon* used in 9grid to present system monitoring information as a network-accessible file hierarchy. The chapter detailed the design and implementation of devmon, the control messages it accepts and the methods for disseminating information that it uses and examined at the performance of devmon and how it may fare in a large cluster or grid environment. Devmon was also compared with other system monitoring software solutions. Chapter 3 also examined how devmon interacts with the hardware monitoring device Winbond W83627THF, which is found on the University of Calgary 9grid computers and used to monitor the cpu fan speeds and temperature sensors on Pentium IV motherboards.

Chapter 4 introduced *resfs*, a resource discovery system created for and used in 9grid. It discussed the tools already existing in Plan 9 to provide similar features in the case of a single installation and how they are expanded and integrated with resfs into a tool useful for the vastly distributed resources utilized in grid computing environments. It also discusses the main features built into resfs and how they can be used to program the output data it provides into a personalized view of the grid's resources. The chapter also described the implementation of resfs in the C language using programming tools available on the Plan 9 operating system. It also provided performance measurements and evaluation of resfs.

5.2 The future of 9grid

Using Plan 9 requires a departure from the familiar set of UNIX paradigms, but with its radical design it facilitates network computing at a level other systems are unable to achieve.

Several obstacles to Plan 9's widespread adoption have hindered its growth in the past. It had a closed-source development model for the first ten years of its existence and some prohibitive restrictions in its initial open source license. These issues have now been resolved so there is no reason why the research community shouldn't investigate this OS as a possible solution for building computational grids.

Although Plan 9 was designed to work in a single administrative domain, the simplicity, clarity and generality of its model allow it to be extended to the inter-organizational world of grid computing. This is easier than adapting a set of systems originally created for a timeshared environment.

9grid shouldn't be looked at as a replacement for grid toolkits, instead it should be examined in its native environment and its design decisions should be taken into consideration when building future grids. 9grid proponents would like to see environments connecting together around a unified lightweight protocol such as 9P and the ability to simplify the creation of grid services to the level of Plan 9, where to create a grid service one only needs to export the name space in which it presents its files. I believe this will create a simpler and more secure grid computing environment.

9grid and related tools should be thought of as an attempt to simplify grids. To reduce the bloat of legacy systems and connect resources in a manner obvious to the user. It is important to simplify grid environments by extending tools the user is already familiar with, instead of introducing a new set of commands which supposedly "do everything right". Today's UNIX systems come with a host of commands which do the same thing but over slightly different communication channels. Example of that is copying files. There are at least three different "simple" programs for copying files: `cp`, `rcp` and `scp`. The first one works at the computer level, copying files between locally mounted partitions. The second one works by copying files across networks between computers running the `r*` dæmons. `Scp` also works across the network, but transports the data securely, using encryption. Each has a set of options incompatible with the other two. Each one has to exist on every UNIX system lest it becomes unusable.

In comparison, the Plan 9 operating system offers only one command, `cp`, which simply creates a copy of a file. It doesn't require knowledge of whether the source or the destination are local or a network away because all communication peculiarities are handled by the `mount` drivers and the 9P protocol.

5.3 Future Work

There are many ways in which the 9grid environment can be extended. Listed below are the ones considered more important.

The most important change being considered for the future of ResFS lies with scalability. Large scale centralized resource monitoring quickly reaches limits imposed by network bandwidth and latency. Other systems such as SuperMon have determined that the frequency of data acquisition is inversely proportional to the number of nodes being monitored [36]. While ResFS is able to access a single data point on a large number of machines simultaneously without incurring the penalty of transferring all other data, it will not perform so well when queried about all data from all nodes, mainly because the network connection is the first thing to get saturated.

A solution to this problem could be an event-driven monitoring model, in which nodes report when information changes and not when they're queried. Thus most events could be monitored infrequently, say 0.1Hz for a load average, while frequent events such as interrupt counts can be queried with a fine-grained minimum frequency, say 100Hz. This frees up network resources from the main monitor and gives the ability to fine-tune the node monitoring hardware to optimize it for a specific environment.

Workflow management is also important in grid environments. Due to the massive constraints imposed by network bandwidth and latency 9grid will not be able to simply "bring all necessary services to the terminal" by connecting to them remotely. Instead, the data needs to be moved closer to the computational resource, with the results collected at a storage resource for later retrieval. This is a worst-case scenario, but it's not inconceivable.

The sum of all those operations is called a *Workflow* and the program that arranges and organizes them is called a *workflow manager* or *workflow scheduler*. Currently a large part of all workflow is being managed by the person who submits the jobs whether directly, by copying data and results around before and after the job's completion, or indirectly, by adding the workflow to the job on a per-job basis. A difficult to optimize task that requires lots of resources.

UNIX systems have an ingenious device that implements workflow management in such an elegant and simple manner that it has become an indispensable part of the system, and one that is being used by everybody without a prerequisite training: the UNIX pipe.

One of the more revolutionary results that came out of UNIX pipes is the toolkit approach to OS design. Building complex systems out of simple, small tools that do one thing but do it well. This technique has been all but forgotten, seeing as how all grid middleware toolkits are built from large, monolithic components interacting through poorly designed, complex protocols. That complexity increases as new functionality is added atop the middleware components. The opposition to exploring new techniques because the current ones "work already" also stifles research in the area. The question is how to achieve the simplicity of the UNIX pipe in a networked environment with the same elegance. The answer is a Workflow Scheduler that integrates grid services seamlessly without being too complex to operate by the average user, based on communicating data between services by reading and writing to files imported over networks.

Processes and system information are increasingly being migrated towards /proc on UNIX and Linux systems. This allows for simpler resource monitoring and management tools to be written, primarily because now they can utilize a simple file-based interface instead of having to be written as kernel modules, or worse, be compiled into the kernel. Nowadays a program needs only to open a file, read it and parse the information to have a complete view of the system. This could be a difficult task all by itself, for example data format in /proc changes very frequently. The difficult task is extending the paradigm across the single system architecture.

A distributed /proc therefore includes a method for combining information from several systems together in a hierarchical name space. It will allow tools to control processes running on several computers at the same time, preferably without modification.

The most important difficulty lies with the fact that all nodes have their own localized process id space and each new process' id number is chosen by the kernel at runtime. It is a daunting task to try and synchronize all the different pid spaces on a network where nodes can number in the thousands, but there may be a solution to this problem by appending node membership information to the process' name.

Authentication in grids is a problem that still hasn't found a simple and elegant non-centralized solution. I discussed briefly what may be required to extend the Plan 9 authentication model to work across administrative domains [52], however we still haven't been able to complete the work for it.

Currently the 9P protocol, the Plan 9 operating system and 9grid do not provide any means of data caching. In other operating systems and grid toolkits caching policy is either unimplemented or is global, i.e., affecting all users at the same time. 9grid, with its private name spaces and the 9P protocol is able to provide a solution to this by extending caching policy through the mount devices on a per-user, user-controllable basis. The net effect is that data caching will be chosen to be the most appropriate for the current application instead of being global for all jobs.

Appendix A

Source Code Listings

A.1 devmon -- hardware and OS monitoring tool

A.1.1 devmon.c – Kernel Device Driver Source

```
#include    "u.h"
#include    "../port/lib.h"
#include    "mem.h"
#include    "dat.h"
#include    "fns.h"
#include    "io.h"
#include    "../port/error.h"

enum {
   Qmondir,
    Qmonctl,
    Qmondata,
    Qtemp,
    Qfans,
  };

static Dirtab mondir[] = {
    ".",        {Qmondir, 0, QTDIR}, 0, 0555,
    "monctl",   {Qmonctl, 0, QTFILE}, 0, 0664,
    "mondata",  {Qmondata, 0, QTFILE}, 0, 0444,
    "temp",     {Qtemp, 0, QTFILE}, 0, 0444,
    "fans",     {Qfans, 0, QTFILE}, 0, 0444,
};


enum
{
    CMcs = 0,
    CMintr,
    CMsyscall,
    CMpfault,
    CMtlbfault,
    CMtlbpurge,
    CMload,
    CMinidle,
    CMinintr,
    CMtemp1,
    CMtemp2,
    CMtemp3,
    CMfan1,
    CMfan2,
    CMfan3,
    CMlast,

    /* for devices with the Windbond W83627THF
     * hardware monitor:
     * http://www.winbond.com.tw/c-winbondhtm/partner/PDFresult.asp?Pname=925
     */
    Bank0    = 0,
    Bank1    = 1,
    Bank2    = 2,

    Tidx     = 0x295,        /* index */
    Tdata    = 0x296,        /* data */

    SysT     = 0x27,        /* SYSTIN */
    Fan1     = 0x28,        /* Fan1 */
    Fan2     = 0x29,        /* Fan2 */
    Fan3     = 0x2a,        /* Fan3 */

    Cr       = 0x40,        /* Configuration Register */
    Isr1     = 0x41,
    Isr2     = 0x42,
    SmiM1    = 0x43,        /* Smi Mask Register 1 */
    SmiM2    = 0x44,
    Fdr1     = 0x47,        /* Fan Divisor Register 1 */
    Did      = 0x49,        /* Device ID */
    Fdr2     = 0x4b,
    BankSel  = 0x4e,        /* Bank Select Register */

    CpuTHi   = 0x50,        /* CPUTIN Temperature Sensor Temp, bits 8:1, Bank 1 */
    CpuTLo   = 0x51,         /* CPUTIN Temperature Sensor Temp, bit 0, Bank 1 */
    VTHi     = 0x50,        /* CPUTIN Temperature Sensor Temp, bits 8:1, Bank 1 */
    VTLo     = 0x51,         /* CPUTIN Temperature Sensor Temp, bit 0, Bank 1 */

};

static Cmdtab monctlmsg[] =
{
    CMcs,        "context",     2,
    CMintr,      "interrupts",  2,
    CMsyscall,   "syscalls",    2,
    CMpfault,    "pfault",      2,
    CMtlbfault,  "tlbfault",    2,
    CMtlbpurge,  "tlbpurge",    2,
    CMload,      "load",        2,
    CMinidle,    "inidle",      2,
    CMinintr,    "inintr",      2,
    CMtemp1,     "systemp",     2,
    CMtemp2,     "cputemp",     2,
    CMtemp3,     "vtemp",       2,
    CMfan1,      "mbfan",       2,
    CMfan2,      "cpufan",      2,
    CMfan3,      "pwfan",       2,

};


typedef struct {
    int flags[CMlast];
} Mon;

static Mon mon;
static char output[4096];

void
wbw(int reg, uchar val)
{
    outb(Tidx, reg);
    outb(Tdata, val);
}

uchar
wbr(int reg)
{
    outb(Tidx, reg);
    return inb(Tdata);
}

void
add(char *a, ...)
{
    va_list arg;

    va_start(arg, a);
    vseprint(output+strlen(output), output+sizeof(output), a, arg);
    va_end(arg);
}

int
rstr(ulong off, char *buf, ulong n, char *str)
{
    int size;

    size = strlen(str);
    if(off >= size)
        return 0;
    if(off+n > size)
        n = size-off;
    memmove(buf, str+off, n);
    return n;
}

void
moninit(void)
{
    int i;

    for(i = 0; i < CMlast; i++) {
        mon.flags[i] = 1;
    }

    /* enable the winbond monitor */
    wbw(0x40, 0x3);
}

Walkqid*
monwalk(Chan* c, Chan *nc, char** name, int nname)
{
    return devwalk(c, nc, name, nname, mondir, nelem(mondir), devgen);
}

static int
monstat(Chan* c, uchar* dp, int n)
{
    return devstat(c, dp, n, mondir, nelem(mondir), devgen);
}

static Chan*
monopen(Chan *c, int omode)
{
    return devopen(c, omode, mondir, nelem(mondir), devgen);
}

static void
monclose(Chan*)
{
}

static long
monread(Chan *c, void *va, long n, vlong offset)
{
    Mach *mp;
    int id;

    output[0] = '\0';
```

```
switch((ulong)c->qid.path){
case Qmondir:
    return devdirread(c, va, n, mondir, nelem(mondir), devgen);
case Qmonctl:
    add("(");
    for(id = 0; id < CMlast; id++)
        add("(%s %s)", monctlmsg[id].cmd, mon.flags[id] ? "true" : "false");
    add(")");
    return rstr(offset, va, n, output);
case Qmondata:
    add("(");

    add("(sysname %s)", sysname);

    for(id = 0; id < 32; id++) {
        if(active.machs & (1<<id)) {
            mp = MACHP(id);
            add("(");
            add("(cpu %d)", id);
            if(mon.flags[CMcs])
                add("(%s %d)", monctlmsg[CMcs].cmd, mp->cs);
            if(mon.flags[CMintr])
                add("(%s %d)", monctlmsg[CMintr].cmd, mp->intr);
            if(mon.flags[CMsyscall])
                add("(%s %d)", monctlmsg[CMsyscall].cmd, mp->syscall);
            if(mon.flags[CMpfault])
                add("(%s %d)", monctlmsg[CMpfault].cmd, mp->pfault);
            if(mon.flags[CMtlbfault])
                add("(%s %d)", monctlmsg[CMtlbfault].cmd, mp->tlbfault);
            if(mon.flags[CMtlbpurge])
                add("(%s %d)", monctlmsg[CMtlbpurge].cmd, mp->tlbpurge);
            if(mon.flags[CMload])
                add("(%s %d)", monctlmsg[CMload].cmd, mp->load);
            if(mon.flags[CMinidle])
                add("(%s %d)", monctlmsg[CMinidle].cmd, (mp->perf.avg_inidle*100)/mp->perf.period);
            if(mon.flags[CMinintr])
                add("(%s %d)", monctlmsg[CMinintr].cmd, (mp->perf.avg_inintr*100)/mp->perf.period);

            /* no negative values */
            if(mon.flags[CMtemp1])
                add("(%s %d)", monctlmsg[CMtemp1].cmd, wbr(SysT));
            if(mon.flags[CMtemp2]) {
                wbw(BankSel, (wbr(BankSel) & ~0x7)|(Bank1));
                add("(%s %d.%d)", monctlmsg[CMtemp2].cmd, wbr(CpuTHi), (wbr(CpuTLo)>>7)*5);
            }
            if(mon.flags[CMtemp3]) {
                wbw(BankSel, (wbr(BankSel) & ~0x7)|(Bank2));
                add("(%s %d.%d)", monctlmsg[CMtemp3].cmd, wbr(VTHi), (wbr(VTLo)>>7)*5);
            }

            /* formula: rpm = 1350000 / (count * divisor) */
            /* divisor default is 2 and we'll leave it that way -- not
             * too fast, mostly power supply fans
             */
            if(mon.flags[CMfan1])
                add("(%s %d)", monctlmsg[CMfan1].cmd, 1350000/(wbr(Fan1)*2));
            if(mon.flags[CMfan2])
                add("(%s %d)", monctlmsg[CMfan2].cmd, 1350000/(wbr(Fan2)*2)-2647);
            if(mon.flags[CMfan3])
                add("(%s %d)", monctlmsg[CMfan3].cmd, 1350000/(wbr(Fan3)*2)-2647);
            add(")");
        }
    }
```

```
        add(")\n");
        return rstr(offset, va, n, output);
    case Qtemp:
        wbw(BankSel, (wbr(BankSel) & ~0x7)|(Bank1));
        /* stats.c doesn't like floats */
        add("%d\t%d\n", wbr(CpuTHi), wbr(SysT));
        return rstr(offset, va, n, output);
    case Qfans:
        wbw(BankSel, (wbr(BankSel) & ~0x7)|(Bank1));
        /* stats.c doesn't like floats */
        add("%d\t%d\t%d\n", 1350000/(wbr(Fan1)*2)-2647,
            1350000/(wbr(Fan2)*2)-2647,
            1350000/(wbr(Fan3)*2)-2647);
        return rstr(offset, va, n, output);
    default:
        break;
    }
    error(Egreg);
    return 0;
}

static long
monwrite(Chan *c, void *va, long n, vlong offset)
{
    char *a, buf[256];
    Cmdbuf *cb;
    Cmdtab *ct;

    USED(offset);
    if(n >= sizeof(buf))
        n = sizeof(buf)-1;
    a = va;
    strncpy(buf, a, n);
    buf[n] = 0;

    switch((ulong)c->qid.path){
    case Qmonctl:
        cb = parsecmd(va, n);
        if(waserror()){
            free(cb);
            nexterror();
        }

        ct = lookupcmd(cb, monctlmsg, nelem(monctlmsg));

        if(ct == nil)
            error("unknown command");

        if(!strncmp(cb->f[1], "true", 4))
            mon.flags[ct->index] = 1;
        else if(!strncmp(cb->f[1], "false", 5))
            mon.flags[ct->index] = 0;
        else
            error("argument can only be true or false");

        free(cb);
        poperror();
        return n;        break;
    default:
        error(Egreg);
    }
    return n;
}
```

```
extern Dev mondevtab;

static Chan*
monattach(char *spec)
{
    return devattach(mondevtab.dc, spec);
}

Dev mondevtab = {
    L’’,
    "mon",

    devreset,
    moninit,
    devshutdown,
    monattach,
    monwalk,
    monstat,
    monopen,
    devcreate,
    monclose,
    monread,
    devbread,
    monwrite,
    devbwrite,
    devremove,
    devwstat,
};
```

A.2 ResFS -- Resource Discovery

A.2.1 resfs-th.h – header file

```
enum {
    Qroot = 0,      /* top level directory */
    Qrootbase,
    Qctl = Qrootbase,
    Qcpufreq,
    Qcputype,
    Qload,
    Qmemory,
    Qncpu,
    Qnproc,
    Qnusers,
    Qobjtype,
    Qos,
    Qpolicy,
    Qsexpr,
    Qsysname,
    Qswap,
    Quptime,
    Qmax,

    Qgroupdir,  /* group directories */
    Qgroupbase,
    Qgctl = Qgroupbase,
    Qpending,
    Qrunning,
    Qdone,

    PNamelen = 64,
    PValuelen = 64,
    Npolicy = 20,

    /* stats-related */
    Maxnum=10,
    /* /dev/swap */
    Mem     = 0,
    Maxmem,
    Swap,
    Maxswap,
    /* /dev/sysstats */
    Procno  = 0,
    Context,
    Interrupt,
    Syscall,
    Fault,
    TLBfault,
    TLBpurge,
    Load,
    Idle,
    InIntr,
    /* /net/ether0/stats */
    In      = 0,
    Out,
    Err0,
    Stacksize   = 8192,
    Blocksize = 4096,   /* a memory page is that many bytes */
    Resplen = 8192,

};

typedef struct Ramfile  Ramfile;
struct Ramfile {
    char *data;
    int ndata;
};


typedef struct FFile FFile;
struct FFile {
    ulong   qid;
    ulong   perm;
    char        *path;

    char *(*report)(char *, int);
};

typedef struct RVal RVal;
struct RVal {
    char name[PNamelen];
    char value[PValuelen];
};

typedef struct Machine Machine;
struct Machine
{
    char        *name;
    char        *objtype;

    int         statsfd;
    int         cputypefd;
    int         btimefd;
    int         swapfd;
    int         procfd;
    int         etherfd;
    int         ifstatsfd;
    int         disable;
    int         ncpu;
    int         nusers;

    int             sexpr;

    vlong       freq;
    vlong       ticks;
    long            nproc;

    ulong       devswap[4];
    ulong       devsysstat[10];
    ulong       prevsysstat[10];
    ulong       netetherstats[8];
    ulong       prevetherstats[8];
    ulong       netetherifstats[2];

    RVal        *policy[Npolicy];

    char        cputype[16];
    ulong   cpufreq;

    char        buf[1024];
    char        *bufp;
    char        *ebufp;

};


/* resfs.c */
extern  FFile   *tab[Qmax];
extern  Machine     mach;
extern  ulong   qidgen;
extern  RWLock  l;
extern  int     sleeptime;
extern  int     noupdate;
void usage(void);

/* stats.c */
void        be2vlong(vlong *, uchar *);
int     loadbuf(int *);
int     readnums(int, ulong *, int);
void        initmach(void);
void        update(void);

/* common.c */
void        initresfs(void);

/* policy.c */
RVal        *setpol(char *, char *);
int     addpol(char *, char *);
int     freepol(char *);

/* BUG: to be changed once we know
 * if there's a better way
 */
#define error sysfatal
```

A.2.2 `common.c` – Common Operations

```
#include <u.h>
#include <libc.h>
#include <thread.h>
#include <fcall.h>
#include <9p.h>

#include "resfs-th.h"

/*
 * TODO: convert the string reporting routines so that we know how far we've gone
 * Needs a global counter in ctlreport, and each report() to return the length of the
 * string it's written.
 */

/* ---- Qctl ---- */
static char *
ctlreport(char *s, int sexp)
{
    int i;
    char *tmp;

    tmp = s;
    if(sexp)
        *tmp++ = '(';

    tab[Qsysname]->report(tmp, sexp);
    tmp = s + strlen(s);
    *tmp++ = ' ';

    for(i = Qrootbase; i < Qmax; i++) {

        if(i == Qctl || i == Qsexpr || i == Qsysname)
            continue;

        tab[i]->report(tmp, sexp);
        tmp = s + strlen(s);
        *tmp++ = ' ';
    }
    if(sexp)
        *tmp++ = ')';

    *tmp = '\0';

    return s;
}

static FFile *
ctlinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0664;
    file->qid = Qctl;
    file->path = "ctl";
    file->report = ctlreport;

    return file;
}

/* ---- Qcpufreq ---- */
static char *
```

```
cpufreqreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(cpufreq %uld)", mach.cpufreq);
    else
        snprint(s, Resplen, "cpufreq=%uld", mach.cpufreq);

    return s;
}
static FFile *
cpufreqinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qcpufreq;
    file->path = "cpufreq";
    file->report = cpufreqreport;

    return file;
}

/* ---- Qcputype ---- */

static char *
cputypereport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(cputype %s)", mach.cputype);
    else
        snprint(s, Resplen, "cputype=%s", mach.cputype);

    return s;
}
static FFile *
cputypeinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qcputype;
    file->path = "cputype";
    file->report = cputypereport;

    return file;
}

/* ---- Qload ---- */

static char *
loadreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(load %uld)", mach.devsysstat[Load]*mach.ncpu);
    else
        snprint(s, Resplen, "load=%uld", mach.devsysstat[Load]*mach.ncpu);

    return s;

}
static FFile *
loadinit(void)
```

```
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qload;
    file->path = "load";
    file->report = loadreport;

    return file;
}

/* ---- Qmemory ---- */
static char *
memreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(memused %uld) (memtotal %uld)",
            mach.devswap[Mem]*Blocksize,
            mach.devswap[Maxmem]*Blocksize);
    else
        snprint(s, Resplen, "memused=%uld memtotal=%uld",
            mach.devswap[Mem]*Blocksize,
            mach.devswap[Maxmem]*Blocksize);

    return s;
}
static FFile *
meminit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qmemory;
    file->path = "memory";
    file->report = memreport;

    return file;
}

/* ---- Qncpu ---- */
static char *
ncpureport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(ncpu %d)", mach.ncpu);
    else
        snprint(s, Resplen, "ncpu=%d", mach.ncpu);

    return s;
}
static FFile *
ncpuinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qncpu;
    file->path = "ncpu";
    file->report = ncpureport;

    return file;
}

/* ---- Qnproc ---- */
static char *
nprocreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(nproc %ld)", mach.nproc);
    else
        snprint(s, Resplen, "nproc=%ld", mach.nproc);

    return s;
}
static FFile *
nprocinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qctl;
    file->path = "nproc";
    file->report = nprocreport;

    return file;
}

/* ---- Qnusers ---- */
static char *
nusersreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(nusers %d)", mach.nusers);
    else
        snprint(s, Resplen, "nusers=%d", mach.nusers);

    return s;
}
static FFile *
nusersinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qnusers;
    file->path = "nusers";
    file->report = nusersreport;

    return file;
}

/* ---- Qobjtype ---- */
static char *
objtypereport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(objtype %s)", mach.objtype);
    else
        snprint(s, Resplen, "objtype=%s", mach.objtype);

    return s;
}
static FFile *
```

```
objtypeinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qobjtype;
    file->path = "objtype";
    file->report = objtypereport;

    return file;
}

/* ---- Qos ---- */
static char *
osreport(char *s, int sexp)
{
    if(sexp)
        sprint(s, "(os Plan9)");
    else
        sprint(s, "os=Plan9");

    return s;
}
static FFile *
osinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qos;
    file->path = "os";
    file->report = osreport;

    return file;
}


/* ---- Qpolicy ---- */
static char *
policyreport(char *s, int sexp)
{
    int i;
    char *tmp = s;

    for(i = 0; i < Npolicy; i++) {
        if(mach.policy[i] == nil)
            continue;

        if(sexp) {
            tmp += sprint(tmp, "(%s %s) ", mach.policy[i]->name, mach.policy[i]->value);
        } else {
            tmp += sprint(tmp, "%s=%s ", mach.policy[i]->name, mach.policy[i]->value);
        }

    }

    return s;
}
static FFile *
policyinit(void)
{
    FFile *file;
    file = emalloc9p(sizeof(FFile));
    file->perm = 0664;
    file->qid = Qpolicy;
    file->path = "policy";
    file->report = policyreport;

    return file;
}

/* ---- Qsexpr ---- */
static char *
sexprreport(char *s, int sexp)
{
    USED(sexp);
    ctlreport(s, 1);
    return s;
}
static FFile *
sexprinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qsexpr;
    file->path = "sexpr";
    file->report = sexprreport;

    return file;
}


/* ---- Qsysname ---- */
static char *
sysreport(char *s, int sexp)
{
    if(sexp)
        sprint(s, "(sys %s)", mach.name);
    else
        sprint(s, "sys=%s", mach.name);

    return s;
}
static FFile *
sysinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qsysname;
    file->path = "sysname";
    file->report = sysreport;

    return file;
}

/* ---- Qswap ---- */
static char *
swapreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(swapused %uld) (swaptotal %uld)",
            mach.devswap[Swap]*Blocksize,
            mach.devswap[Maxswap]*Blocksize);
    else
        snprint(s, Resplen, "swapused=%uld swaptotal=%uld",
            mach.devswap[Swap]*Blocksize,
            mach.devswap[Maxswap]*Blocksize);

    return s;
}
static FFile *
swapinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0444;
    file->qid = Qswap;
    file->path = "swap";
    file->report = swapreport;

    return file;
}

/* ---- Quptime ---- */
static char *
upreport(char *s, int sexp)
{
    if(sexp)
        snprint(s, Resplen, "(up %lldd:%2.2lldh:%2.2lldm:%2.2llds)",
            (mach.ticks/mach.freq)/86400,
            ((mach.ticks/mach.freq)%86400)/3600,
            ((mach.ticks/mach.freq)%3600)/60,
            ((mach.ticks/mach.freq)%60));
    else
        snprint(s, Resplen, "up=%lldd:%2.2lldh:%2.2lldm:%2.2llds",
            (mach.ticks/mach.freq)/86400,
            ((mach.ticks/mach.freq)%86400)/3600,
            ((mach.ticks/mach.freq)%3600)/60,
            ((mach.ticks/mach.freq)%60));
    return s;
}
static FFile *
upinit(void)
{
    FFile *file;

    file = emalloc9p(sizeof(FFile));
    file->perm = 0664;
    file->qid = Qctl;
    file->path = "uptime";
    file->report = upreport;

    return file;
}


void
initresfs(void)
{
    tab[Qctl] = ctlinit();
    tab[Qcpufreq] = cpufreqinit();
    tab[Qcputype] = cputypeinit();
    tab[Qload] = loadinit();
    tab[Qmemory] = meminit();
    tab[Qncpu] = ncpuinit();
    tab[Qnproc] = nprocinit();
    tab[Qnusers] = nusersinit();
    tab[Qobjtype] = objtypeinit();
    tab[Qos] = osinit();
    tab[Qpolicy] = policyinit();    /* to be added later */
    tab[Qsexpr] = sexprinit();      /* is called through Qctl */
    tab[Qsysname] = sysinit();
    tab[Qswap] = swapinit();
    tab[Quptime] = upinit();
}
```

A.2.3 `policy.c` – Policy Implementation

```
/*
 * helper functions for policy reporting
 *
 */
#include <u.h>
#include <libc.h>
#include <thread.h>
#include <fcall.h>
#include <9p.h>
#include <ctype.h>

#include "resfs-th.h"


static RVal *
findpol(char *name)
{
    int i;

    if(name == nil)
        return nil;

    for(i = 0; i < Npolicy; i++) {
        if(mach.policy[i] == nil)
            continue;
        if(!strncmp(mach.policy[i]->name, name, PNamelen))
            return mach.policy[i];
    }

    return nil;
}

RVal *
setpol(char *name, char *value)
{
    RVal *r;

    r = findpol(name);
    if(r == nil)
        return nil;

    strncpy(r->value, value, PValuelen);
    return r;
}

int
addpol(char *name, char *value)
{
    int i;
    RVal *r;

    for(i = 0; i < Npolicy; i++)
        if(mach.policy[i] == nil)
            break;

    if(i == Npolicy)
        return -1;

    r = emalloc9p(sizeof(RVal));
    if(r == nil)
        return -1;

    strncpy(r->name, name, PNamelen);
    strncpy(r->value, value, PValuelen);

    mach.policy[i] = r;

    return i;
}

int
freepol(char *name)
{
    int i;

    for(i = 0; i < Npolicy; i++) {
        if(mach.policy[i] == nil)
            continue;
        if(strncmp(mach.policy[i]->name, name, PNamelen) == 0)
            break;
    }
    if(i == Npolicy)
        return -1;

    free(mach.policy[i]);
    mach.policy[i] = nil;

    return i;
}
```

A.2.4 `resfs-th.c` – Main Threaded Code

```
#include <u.h>
#include <libc.h>
#include <thread.h>
#include <fcall.h>
#include <9p.h>
#include <ctype.h>
#include <bio.h>
#include <ndb.h>

#include "resfs-th.h"

FFile   *tab[Qmax];
Tree        *restree;
RWLock  l;
int         sleeptime = 1000;
int         debug;
int     noupdate;   /* do not update info periodically */
int     nolisten;   /* do not listen on 18000 */
int     nopost; /* do not post to $resourcefs */
Machine     mach;

void
usage(void)
{
    fprint(2, "usage: resfs [-qpu] [-s service] [-t sleeptime]\n");
    sysfatal("usage");
}

void
fscreate(Req *r)
{
    File *f;
    int t;

    t = (int)r->fid->file->aux;

    if(t <= Qmax) {
        respond(r, "creating an already existing file?");
        return;
    }

    if(t == Qroot) {
        if(f = createfile(r->fid->file, r->ifcall.name, r->fid->uid, r->ifcall.perm, nil)){
//          rf = emalloc9p(sizeof *rf);
            f->aux = (void *)(Qgroupdir);
            r->fid->file = f;
            r->ofcall.qid = f->qid;
            respond(r, nil);
            return;
        }
    }
    respond(r, "could not create file");
}

void
fsread(Req *r)
{
    int t;
    char tmp[Resplen];

    t = (int)r->fid->file->aux;

    if(t > Qmax)
        respond(r, "read from unknown file");

    if(tab[t]->report != nil) {
        rlock(&l);
        tab[t]->report(tmp, mach.sexpr);
        *(strchr(tmp, '\0')) = '\n';
        *(strchr(tmp, '\n')+1) = '\0';
        runlock(&l);
    }
    readstr(r, tmp);
    respond(r, nil);
}

void
fswrite(Req *r)
{
    char tmp[Resplen];
    char *args[3];
    int t;
```

```
	t = (int)r->fid->file->aux;

	switch(t){
	default:
		if(t > Qmax)
			respond(r, "write to unknown file");
		else {
			sprint(tmp, "write to %s not supported", tab[t]->path);
			respond(r, tmp);
		}
		return;
	case Qpolicy:
		if(r->ifcall.count >= sizeof tmp){
			respond(r, "write too long");
			return;
		}
		memmove(tmp, r->ifcall.data, r->ifcall.count);
		tmp[r->ifcall.count] = '\0';

		t = tokenize(tmp, args, 3);
		if(t  > 3) {
			respond(r, "too many arguments (expected 3)");
			return;
		}

		if(cistrncmp(args[0], "add", 3) == 0) {
			if(t != 3) {
				respond(r, "not enough arguments for 'add'");
				return;
			}
			if(addpol(args[1], args[2]) < 0) {
				sprint(tmp, "%s: cannot add policy", args[1]);
				respond(r, tmp);
				return;
			}
		} else if(cistrncmp(args[0], "set", 3) == 0) {
			if(t != 3) {
				respond(r, "not enough arguments for 'set'");
				return;
			}
			if(setpol(args[1], args[2]) == nil) {
				sprint(tmp, "%s: policy not found", args[1]);
				respond(r, tmp);
				return;
			}
		} else if(cistrncmp(args[0], "del", 3) == 0) {
			if(t != 2) {
				respond(r, "too many arguments for 'del'");
				return;
			}
			if(freepol(args[1]) < 0) {
				sprint(tmp, "%s: policy not found", args[1]);
				respond(r, tmp);
				return;
			}
		} else {
			respond(r, "unknown command");
			return;
		}
		break;

	case Qctl:
		if(r->ifcall.count >= sizeof tmp){
```

```
            respond(r, "write too long");
            return;
        }
        memmove(tmp, r->ifcall.data, r->ifcall.count);
        tmp[r->ifcall.count] = '\0';

        t = tokenize(tmp, args, 3);
        if(t  > 3) {
            respond(r, "too many arguments (expected 3)");
            return;
        }

        if(cistrncmp(args[0], "set", 2) == 0) {
            if(t != 2) {
                respond(r, "not enough arguments for 'set'");
                return;
            }
            if(cistrncmp(args[1], "sexpr", 5) == 0) {
                mach.sexpr = 1;
            } else if(cistrncmp(args[1], "nosexpr", 5) == 0) {
                mach.sexpr = 0;
            } else {
                sprint(tmp, "unknown argument: %s", args[1]);
                respond(r, tmp);
                return;
            }
        }
        break;
    }

    r->ofcall.count = r->ifcall.count;
    respond(r, nil);
    return;
}

void
fscleanup(Srv *s)
{
    USED(s);
    exits(nil);
}


Srv fs=
{
    .read       =   fsread,
    .write  =   fswrite,
    .create =   fscreate,
};

/* Find out where aggregate is (resourcefs in ndb)
 * and post a file descriptor there.
 */
void
callagg(void)
{
    Srv *s;
    int fd;

    switch(rfork(RFPROC|RFMEM|RFNOWAIT)){
    case -1:
        sysfatal("rfork: %r");
    default:
        return;
    case 0:
        break;
    }

    fd = dial(netmkaddr("$resourcefs", 0, "18001"), 0, 0, 0);
    if(fd < 0)
        exits("dial: %r\n");

    print("dialed resfs...\n");
    s = emalloc9p(sizeof *s);
    *s = fs;
    s->infd = s->outfd = fd;
    srv(s);
    free(s);
    exits(nil);
}

void
listensrv(void)
{
    Srv *s;
    char adir[NETPATHLEN], ldir[NETPATHLEN];
    int acfd, dfd, lcfd;

    switch(rfork(RFPROC|RFMEM|RFNOWAIT)){
    case -1:
        sysfatal("rfork: %r");
    default:
        return;
    case 0:
        break;
    }

    acfd = announce("tcp!*!18000", adir);
    if(acfd < 0)
        sysfatal("announce: %r");

    for(;;){
        lcfd = listen(adir, ldir);
        if(lcfd < 0)
            sysfatal("listen: %r");

        switch(rfork(RFPROC | RFMEM | RFNOWAIT)){
        case -1:
            fprint(2, "fork listen: %r");
            continue;
        default:
            continue;
        case 0:
            break;
        }
        dfd = accept(lcfd, ldir);
        if(dfd < 0)
            sysfatal("accept: %r");

        s = emalloc9p(sizeof *s);
        *s = fs;
        s->infd = s->outfd = dfd;
        srv(s);
        free(s);
        _exits(nil);
    }
}
```

```
void
main(int argc, char **argv)
{
    File *f, *rootf;
    int i;
    char *sysn;
    char *service = "resfs";

    ARGBEGIN{
    case ’D’:
        chatty9p++;
        break;
    case ’d’:
        debug++;
        break;
    case ’s’:
        service = EARGF(usage());
        break;
    case ’p’:
        nopost++;
        break;
    case ’u’:
        noupdate++;
        break;
    case ’q’:
        nolisten++;
        break;
    case ’t’:
        sleeptime = atoi(EARGF(usage()));
        break;
    default:
        usage();
    }ARGEND

    if(argc > 0)
        usage();

    USED(service);

    sysn = sysname();
    if(! sysn)
        sysn = "unknown";

    initmach();
    initresfs();

    restree = fs.tree = alloctree(getuser(), getuser(), DMDIR|0555, nil);
    incref(restree->root);
    rootf = createfile(restree->root, sysn, getuser(), DMDIR|0775, nil);
    if(rootf == nil)
        sysfatal("creating %s: %r", sysn);
    rootf->aux = (void *)Qroot;

    for(i = Qrootbase; i < Qmax; i++) {
        f = createfile(rootf, tab[i]->path, getuser(), tab[i]->perm, nil);
        if(f == nil)
            sysfatal("creating %s: %r", tab[i]->path);
        f->aux = (void *)i;
    }
    decref(restree->root);


    if(!nolisten)
        listensrv();
    if(!nopost)
        callagg();

    update();


//  postmountsrv(&fs, service, nil, 0);


    exits(nil);
}
```

A.2.5 `stats-th.c` – Statistics and Monitoring

```
/*
 * helper functions for stats collection
 *
 * lifted from stats(1) mostly
 */
#include <u.h>
#include <libc.h>
#include <thread.h>
#include <fcall.h>
#include <9p.h>
#include <ctype.h>

#include "resfs-th.h"

static uvlong order = 0x0001020304050607ULL;

void
be2vlong(vlong *to, uchar *f)
{
    uchar *t, *o;
    int i;

    t = (uchar*)to;
    o = (uchar*)&order;
    for(i = 0; i < 8; i++)
        t[o[i]] = f[i];
}

int
loadbuf(int *fd)
{
    int n;

    if(*fd < 0)
        return 0;
    seek(*fd, 0, 0);
    n = read(*fd, mach.buf, sizeof mach.buf);
    if(n <= 0){
        close(*fd);
        *fd = -1;
        return 0;
    }
    mach.bufp = mach.buf;
    mach.ebufp = mach.buf+n;
    return 1;
}

int
```

```
readnums(int n, ulong *a, int spanlines)
{
    int i;
    char *p, *ep;

    if(spanlines)
        ep = mach.ebufp;
    else
        for(ep=mach.bufp; ep<mach.ebufp; ep++)
            if(*ep == '\n')
                break;
    p = mach.bufp;
    for(i=0; i<n && p<ep; i++){
        while(p<ep && !isdigit(*p) && *p!='-')
            p++;
        if(p == ep)
            break;
        a[i] = strtoul(p, &p, 10);
    }
    if(ep < mach.ebufp)
        ep++;
    mach.bufp = ep;
    return i == n;
}

void
initmach(void)
{
    char buf[256];

    mach.name = sysname();
    if(mach.name == nil)
        mach.name = "unknown";
    mach.objtype = getenv("objtype");
    if(mach.objtype == nil)
        mach.objtype = "unknown";

    snprint(buf, sizeof buf, "#c/swap");
    mach.swapfd = open(buf, OREAD);
    memset(mach.devswap, 0, sizeof mach.devswap);

    snprint(buf, sizeof buf, "#c/sysstat");
    mach.statsfd = open(buf, OREAD);
    memset(mach.devsysstat, 0, sizeof mach.devsysstat);

    snprint(buf, sizeof buf, "#P/cputype");
    mach.cputypefd = open(buf, OREAD);
    memset(mach.cputype, 0, sizeof mach.cputype);
    mach.cpufreq = 0;

    /* cpufreq can change without rebooting on laptops
     * #P won't pick up the change though.
     */
    if(loadbuf(&mach.cputypefd)) {
        mach.cpufreq = atoi(strrchr(mach.buf, ' ')+1);
        *(strchr(mach.buf, ' ')) = '\0';
        strncpy(mach.cputype, mach.buf, sizeof mach.cputype - 1);
    }

    snprint(buf, sizeof buf, "/proc");
    mach.procfd = open(buf, OREAD);
    mach.nproc = 0;

    snprint(buf, sizeof buf, "#c/bintime");
    mach.btimefd = open(buf, OREAD);
    mach.freq = 0;
    mach.ticks = 0;

    mach.ncpu = 0;
    mach.nusers = 0;

    mach.sexpr = 0;

    addpol("maxproc", "1000");
    addpol("maxnode", "5");
    addpol("maxjob", "50");
    addpol("maxwalltime", "inf");
    addpol("maxmem", "5");
    addpol("maxload", "10000");
}


void
update(void)
{
    ulong a[Maxnum];
    char b[24];
    Dir *dir;
    int n, i;
    vlong t;

    switch(rfork(RFPROC|RFMEM|RFNOWAIT)){
    case -1:
        sysfatal("rfork: %r");
    default:
        return;
    case 0:
        break;
    }

    for(;;) {
        wlock(&l);

        if(loadbuf(&mach.swapfd) && readnums(nelem(mach.devswap), a, 0))
            memmove(mach.devswap, a, sizeof mach.devswap);
        else
            mach.devswap[Maxmem] = mach.devswap[Maxswap] = 100;

        if(loadbuf(&mach.statsfd)){
            memset(mach.devsysstat, 0, sizeof mach.devsysstat);
            for(n=0; readnums(nelem(mach.devsysstat), a, 0); n++)
                for(i=0; i<nelem(mach.devsysstat); i++)
                    mach.devsysstat[i] += a[i];
            mach.ncpu = n;
        }


        if(read(mach.btimefd, b, 24) == 24) {
            be2vlong(&t, (uchar*)b+8);
            mach.ticks = t;
            be2vlong(&t, (uchar*)b+16);
            mach.freq = t;
        }

        seek(mach.procfd, 0, 0);
        mach.nproc = dirreadall(mach.procfd, &dir);

        wunlock(&l);
        /* clean up */
        free(dir);

        if(noupdate)
            exits(nil);

        sleep(sleeptime);
    }
}
```

A.3 AggrFS -- ResFS Aggregate Nodes

```
#include <u.h>
#include <libc.h>
#include <fcall.h>
#include <thread.h>
#include <9p.h>
#include <bio.h>
#include <ndb.h>

static int debug;
#define DEBUG if(!debug){} else fprint

char *service = "aggrfs";
char *progname;

static void
usage(void)
{
    fprint(2, "aggrfs [-dD] [-s srvname]\n");
    exits("usage");
}



void
listener(void)
{
    char adir[NETPATHLEN], ldir[NETPATHLEN];
    int acfd, dfd, lcfd;

    switch(rfork(RFPROC|RFMEM|RFNOWAIT)){
    case -1:
        sysfatal("rfork: %r");
    default:
        return;
    case 0:
        break;
    }


    acfd = announce("tcp!*!18001", adir);
    if(acfd < 0)
        sysfatal("announce: %r");

    for(;;){
        lcfd = listen(adir, ldir);
        if(lcfd < 0)
            sysfatal("listen: %r");
        dfd = accept(lcfd, ldir);
        if(dfd < 0)
            sysfatal("accept: %r");


        mount(dfd, -1, "/tmp", MAFTER, "");
    }
}

void
listensrv(void)
{
    Srv *s;
    char adir[NETPATHLEN], ldir[NETPATHLEN];
    int acfd, dfd, lcfd;
    char *arglist[7], **argp;

    switch(rfork(RFPROC|RFMEM|RFNOWAIT)){
    case -1:
        sysfatal("rfork: %r");
    default:
        return;
    case 0:
        break;
    }

    acfd = announce("tcp!*!18000", adir);
    if(acfd < 0)
        sysfatal("announce: %r");

    for(;;){
        lcfd = listen(adir, ldir);
        if(lcfd < 0)
            sysfatal("listensrv: listen: %r");

        switch(rfork(RFPROC | RFMEM | RFNOWAIT | RFCFDG)){
        case -1:
            fprint(2, "fork listensrv: %r");
            continue;
        default:
            continue;
        case 0:
            break;
        }
        dfd = accept(lcfd, ldir);
        if(dfd < 0)
            sysfatal("listensrv: accept: %r");
        DEBUG(2, "listensrv: accepted fd: %d; ldir: %s\n", dfd, ldir);

        argp = arglist;

        *argp++ = progname;
        *argp++ = "-r";
        *argp++ = "/tmp";
        *argp = nil;

        dup(dfd, 0);

        DEBUG(2, "listensrv: exec\n");
        exec("/bin/exportfs", arglist);
        fprint(2, "can't exec exportfs: %r\n");
        exits("exec");
    }
}

static void
fsread(Req *r)
{
    respond(r, nil);
}

Srv fs=
{
    .read       =   fsread,
};

void
main(int argc, char **argv)
{
    File *f;

    progname = argv[0];

    ARGBEGIN{
    case 'D':
        chatty9p++;
        break;
    case 'd':
        debug++;
        break;
    case 's':
        service = EARGF(usage());
        break;
    default:
        usage();
    }ARGEND

    if(argc > 1)
        usage();

    fs.tree = alloctree("sys", "sys", DMDIR|0555, nil);
    incref(fs.tree->root);
    f = createfile(fs.tree->root, "ctl", getuser(),  0664, nil);
    if(f == nil)
        sysfatal("creating ctl: %r");
    f->aux = nil;
    decref(fs.tree->root);

    postmountsrv(&fs, service, "/tmp", MREPL);

    listener();
    listensrv();

    exits(nil);
}
```

A.4 Mach – Per-processor Machine Definition

```
struct Mach
{
    int      machno;           /* physical id of processor (KNOWN TO ASSEMBLY) */
    ulong    splpc;            /* pc of last caller to splhi */

    ulong*   pdb;              /* page directory base for this processor (va) */
    Tss*     tss;              /* tss for this processor */
    Segdesc* gdt;              /* gdt for this processor */

    Proc*    proc;             /* current process on this processor */
    Proc*    externup;         /* extern register Proc *up */

    Page*    pdbpool;
    int      pdbcnt;

    ulong    ticks;            /* of the clock since boot time */
    Label    sched;            /* scheduler wakeup */
    Lock     alarmlock;        /* access to alarm list */
    void*    alarm;            /* alarms bound to this clock */
    int      inclockintr;

    Proc*    readied;          /* for runproc */
    ulong    schedticks;       /* next forced context switch */

    int      tlbfault;
    int      tlbpurge;
    int      pfault;
    int      cs;
    int      syscall;
    int      load;
    int      intr;
    int      flushmmu;         /* make current proc flush it's mmu state */
    int      ilockdepth;
    Perf     perf;             /* performance counters */

    ulong    spuriousintr;
    int      lastintr;

    int      loopconst;

    Lock     apictimerlock;
    int      cpumhz;
    uvlong   cyclefreq;        /* Frequency of user readable cycle counter */
    uvlong   cpuhz;
    int      cpuidax;
    int      cpuiddx;
    char     cpuidid[16];
    char*    cpuidtype;
    int      havetsc;
    int      havepge;
    uvlong   tscticks;

    vlong    mtrrcap;
    vlong    mtrrdef;
    vlong    mtrrfix[11];
    vlong    mtrrvar[32];      /* 256 max. */

    int    stack[1];
};
```

Bibliography

[1] T. L. Sterling, J. Salmon, D. J. Becker, D. F. Savarese: *How to Build a Beowulf*, The MPI Press, 1999.

[2] G. Bell, J Grey: *What's next in high performance computing?* Communications of the ACM, vol 45, issue 2, 2002.

[3] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman: *Grid Service Specification, Draft 3*, Global Grid Forum, July 2002.

[4] I. Foster, C. Kesselman, S. Tuecke: *The Anatomy of the Grid: Enabling scalable virtual organizations*, International J. Supercomputer Applications, 15(3), 2001.

[5] J. Nick I. Foster, C. Kesselman, S. Tuecke: *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[6] H. Trickey: *APE - The ANSI/POSIX Environment*, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[7] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom: *Plan 9 from Bell Labs*, Computing Systems, 8(3):221-254, 1995.

[8] D. Presotto, P. Winterbottom: *The IL Protocol*, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 2000.

[9] S. Mullender, D. Presotto: *Programming Distributed Applications using Plan 9 from Bell Labs*, Proceedings of the European Research Seminar on Advances in Distributed Systems, Bertinoro, Italy, May, 2001.

[10] R. Pike, K. Thompson: *Hello World*, Proceedings of the Winter 1993 USENIX Conference, 43-50, San Diego, 1993.

[11] R. Minnich: *Private Namespaces for Linux*, Dr. Dobb's Journal, Dec 2001.

[12] R. Cox, E. Grosse, R. Pike, D. Presotto, S. Quinlan: *Security in Plan 9*, Proceedings of the 11th USENIX Security Symposium, pp. 3–16, 2002.

[13] R. Pike, D. Presotto, K. Thompson, H. Trickey, P. Winterbottom: *The Use of Name Spaces in Plan 9*, Op. Sys. Rev., Vol. 27, No. 2, April 1993, pp. 72-76.

[14] J. Novotny, S. Tuecke, V. Welch: *An Online Credential Repository for the Grid: MyProxy. Proceedings of the Tenth International*, Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[15] Globus Alliance: *The Globus Project*, `http://www.globus.org`.

[16] Globus Alliance: *GridFTP - Universal Data Transfer for the Grid*, White Paper. September 5, 2000.

[17] A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, A. S. Grimshaw: *The Legion Grid Portal*, Concurrency and Computation: Practice and Experience Vol. 14, Grid Computing environments Special Issue 13-14, 2002.

[18] The Globus Alliance: *The WS-Resource Framework*,
http://www.globus.org/wsrf

[19] I. Foster, N. Karonis, C. Kesselman, G. Koenig, S. Tuecke: *A Secure Communications Infrastructure for High-Performance Distributed Computing*, 6th IEEE Symp. on High-Performance Distributed Computing, pp. 125-136, 1997.

[20] I. Foster, S. Tuecke: *Enabling Technologies for Web-Based Ubiquitous Supercomputing*, Proc. 5th IEEE Symp. on High Performance Distributed Computing, pp. 112-119, 1996.

[21] I. Foster, C. Kesselman, S. Tuecke: *The Nexus Task-Parallel Runtime System*, Proc. 1st Int'l Workshop on Parallel Processing, pp. 457-462, 1994.

[22] D. Dullmann, W. Hoschek, J. Jean-Martinez, A. Samar, H. Stockinger, K. Stockinger: *Models for Replica Synchronisation and Consistency in a Data Grid*, 10th IEEE Symposium on High Performance and Distributed Computing

[23] K. Ranganathan, Adriana Iamnitchi, and I. Foster: *Improving Data Availability through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities*, Proceedings of Global and Peer-to-Peer Computing on Large Scale Distributed Systems Workshop, Berlin, Germany, May 2002.

[24] S. Quinlan,S. Dorward: *Venti: a new approach to archival storage*, Conference on File and Storage Technologies, Monterey, CA, 28–30 January 2002.

[25] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, S. Tuecke: *Security for Grid Services*, Twelfth International Symposium on High Performance Distributed Computing (HPDC-12), IEEE Press, June 2003.

[26] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling: *Open Grid Services Infrastructure (OGSI) Version 1.0*, Global Grid Forum Draft Recommendation, 6/27/2003.

[27] R. Figueiredo, P. Dinda, J. Fortes: *A Case for Grid Computing on Virtual Machines*, In Proceedings of the Intlernational Conference on Distributed Computing Systems (ICDCS), 04/2003.

[28] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, S. Tuecke: *A Directory Service for Configuring High-Performance Distributed Computations*, Proc. 6th IEEE Symposium on High-Performance Distributed Computing, pp. 365-375, 1997.

[29] The OpenPBS Project: *http://www.openpbs.org*.

[30] The Condor Project: *http://www.cs.wisc.edu/condor/*.

[31] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch: *The Sprite network operating system*, IEEE Computer, 21(2):23–36, February 1988.

[32] S.J. Mullender, G. Van Rossum, A.S. Tanenbaum, R. Van Renesse, H. Van Staveren: *Amoeba: A distributed operating system for the 1990s*, IEEE Computer, 14:365–368, May 1990.

[33] I. Foster, G. von Laszewski: *Usage of LDAP in Globus*. `http://www.globus.org`.

[34] Ron Minnich, private communication, June 2003.

[35] Rob Simmonds, private communication, August 2004.

[36] M. Sottile, R. Minnich: *"Supermon: A High-Speed Cluster Monitoring System"*, IEEE Conference on Cluster Computing, September 24-26 2002.

[37] SuperMon: *http://sexpr.sourceforge.net/*.

[38] J. McCarthy et al.: *LISP 1.5 Programmer's Manual, 2nd edition*, MIT Press, 1965.

[39] Phil Winterbottom: *Alef Language Reference Manual*, Plan 9 Programmer's Manual, Volume 2, AT&T Bell Laboratories, Murray Hill, NJ, 1995.

[40] *Introduction to the 9p protocol*, Plan 9 Programmer's Manual, Volume 3, AT&T Bell Laboratories, Murray Hill, NJ, 2000.

[41] *Distributed and Network Operating Systems*, http://www.csee.wvu.edu/∼jdm/classes/cs258/OScat/distr.html.

[42] *WestGRID: Western Canada Research Grid*, http://www.westgrid.ca.

[43] *TeraGRID: Distributed Infrastructure for Open Scientific Research*, http://www.teragrid.org.

[44] *Reality Grid Project*, http://www.realitygrid.org.

[45] R. H. Thomas, *A Resource Sharing Executive for the ARPANET*, NCC, Volume 42, 1973, pp. 155-163.

[46] E. Strohmaier, J. Dongarra, *Highlights of the 23rd TOP500 List*, International Supercomputing Conference, Heidelberg, Germany, 2004.

[47] Plan 9 from Bell-Labs: *http://plan9.bell-labs.com/plan9dist*.

[48] Vita Nuova Holdings: *http://www.vitanuova.com*.

[49] Winbond W83627THF datasheet: *http://www.winbond.com.tw/c-winbondhtm/partner/PDFresult.asp?Pname=925*.

[50] Lm-Sensors, Linux System Hardware Monitoring: *http://secure.netroedge.com/∼lm78/*.

[51] T. J. Killian: *Processes as Files*, USENIX Summer 1984 Conference Proceedings, June 1984, Salt Lake City, UT.

[52] A. Mirtchovski, R. Simmonds, R. Minnich: *Plan 9 – An Integrated Approach To Grid Computing*, IPDPS-04 April 26-30 2004, Santa Fe, NM, USA

[53] W. R. Stevens: *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992, ISBN 0-201-56317-7.

[54] J. Chin, P. V. Coveney: *Towards tractable toolkits for the Grud: a plea for lightweight, usable middleware* UK e-Science Technical Report, number UKeS-2004-01

[55] IBM DeveloperWorks: *Web Services Notification and Web Services Resource Framework*,
`http://www.ibm.com/developerworks/webservices/library/ws-resource`

[56] R. J. Allan , D. R. S. Boyd , T. Folkes , C. Greenough , D. Hanlon , R. P. Middleton , R. A. Sansum: *Evaluation of Globus and Associated Grid Middleware* CLRC e-Science Centre, 2001

[57] B. Butchart , C. Chapman , W. Emmerich: *OGSA First Impressions: A Case Study using the Open Grid Service Architecture*, Proceedings of the UK E-Science All Hands Meeting, Nottingham pp 810-816, 2003

[58] *Plan 9 Software Ported to Unix*,
http://swtch.com/plan9port/

[59] *The 9grid Project*,
http://www.9grid.net

[60] *BigBangwidth*,
http://www.bigbangwidth.com

[61] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, R. Neugebauer: *Xen and the Art of Virtualization*. In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.

[62] *VMWare Virtual Infrastructure*,
http://www.vmware.com

[63] *Big Brother System and Network Monitoring Tools*,
http://www.bb4.org/

[64] *Berkeley DB*,
http://www.sleepycat.com/

[65] *Multi-Router Traffic Grapher*,
http://people.ee.ethz.ch/ oetiker/webtools/mrtg/

[66] *The Sysstat System Monitoring Suite*,
http://perso.wanadoo.fr/sebastien.godard/

[67] Clifford Neumann: *The Kerberos Network Authentication Service (V5). Internet Draft ietf-cat-kerb-kerberos-revision-04.txt*, June 1999.

[68] J. Oikarinen, D. Reed: *RFC 1459: Internet Relay Chat Protocol*, May 1993.