

DESIGN AND IMPLEMENTATION OF GRIDOS: OPERATING SYSTEM  
SERVICES FOR GRID ARCHITECTURES

By

PRADEEP PADALA

A THESIS PRESENTED TO THE GRADUATE SCHOOL  
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT  
OF THE REQUIREMENTS FOR THE DEGREE OF  
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

Copyright 2003  
by  
Pradeep Padala

## ACKNOWLEDGMENTS

I would like to gratefully acknowledge the great supervision of Dr. Joseph Wilson during this work. I thank Dr. Michael Frank and Dr. Paul Avery for serving on my committee and for reviewing my work.

I also would like to thank Dr. Richard Cavannaugh and Dr. Sanjay Ranka for their helpful suggestions in grid file system work.

I am grateful to all my friends who helped directly or indirectly in preparing this work.

Finally, I am forever indebted to my parents for helping me to reach this stage in my life.

## TABLE OF CONTENTS

	<u>page</u>
ACKNOWLEDGMENTS . . . . .	iii
LIST OF FIGURES . . . . .	vi
ABSTRACT . . . . .	vii
1 INTRODUCTION . . . . .	1
2 MOTIVATION . . . . .	3
2.1 The Grid . . . . .	3
2.2 Grid Applications . . . . .	4
2.3 Grid Middleware . . . . .	5
2.4 Globus . . . . .	6
2.4.1 Security . . . . .	8
2.4.2 Data Management . . . . .	8
2.4.3 Resource Management . . . . .	9
2.4.4 Information Services . . . . .	9
2.5 Condor . . . . .	10
2.6 Legion . . . . .	12
2.7 Previous Work . . . . .	12
3 GRIDOS DESIGN . . . . .	15
3.1 Core Principles . . . . .	15
3.2 Core Modules . . . . .	16
3.2.1 gridos_io: High-Performance I/O Module . . . . .	17
3.2.2 gridos_comm: Communication Module . . . . .	18
3.2.3 gridos_rm: Resource Management Module . . . . .	20
3.2.4 gridos_pm: Process Management Module . . . . .	20
3.3 Additional Modules . . . . .	21
3.3.1 gridos_ftp_common . . . . .	21
3.3.2 gridos_ftp_server . . . . .	21
3.3.3 gridos_ftp_client . . . . .	21
4 IMPLEMENTATION . . . . .	22
4.1 Linux Kernel Module Architecture . . . . .	22
4.2 Kernel Level Web/Ftp Server (tux) . . . . .	23
4.2.1 User Level Access to kernel HTTP layer . . . . .	24

4.2.2	Listening & Accepting Messages . . . . .	24
4.3	Modules . . . . .	29
4.3.1	Activation . . . . .	29
4.3.2	IO Module . . . . .	29
4.3.3	FTP Client and Server Modules . . . . .	31
4.4	Library Wrapper . . . . .	31
4.5	GridOS Middleware . . . . .	32
5	SCENARIOS . . . . .	33
5.1	Scenario #1: Transporting a file . . . . .	33
5.2	Scenario #2: Reading and Writing to a file locally . . . . .	33
6	PERFORMANCE . . . . .	36
6.1	Test Environment and Methodology . . . . .	36
6.2	GridOS vs Proftpd . . . . .	37
6.3	GridFTP vs GridOS ftp server using <code>globus-url-copy</code> client . . . . .	38
6.4	GridFTP server/client vs GridOS ftp server/client . . . . .	38
7	GRID FILE SYSTEM . . . . .	39
7.1	Motivating Example . . . . .	40
7.2	Requirements . . . . .	43
7.3	Design . . . . .	44
7.3.1	Logical Hierarchical Name Space . . . . .	44
7.3.2	Data Access/Transfer and Replica Management . . . . .	44
7.3.3	POSIX semantics . . . . .	44
7.3.4	Metadata . . . . .	46
7.3.5	Automatic Data Management . . . . .	46
7.3.6	Performance Optimization . . . . .	46
7.4	Building the Service . . . . .	47
7.4.1	ServiceData . . . . .	47
7.4.2	Notification . . . . .	48
7.5	Open Questions . . . . .	48
8	FUTURE WORK . . . . .	49
9	CONCLUSIONS . . . . .	50
	APPENDIX Grid File System GWSDDL Description . . . . .	51
	REFERENCES . . . . .	57
	BIOGRAPHICAL SKETCH . . . . .	60

## LIST OF FIGURES

<u>Figure</u>	<u>page</u>
2-1 Layered grid architecture . . . . .	7
2-2 A ClassAd example . . . . .	11
3-1 Major modules and structure of GridOS . . . . .	16
4-1 Tux actions . . . . .	25
4-2 Connection accept . . . . .	26
4-3 Event loop . . . . .	27
4-4 Accept request . . . . .	28
4-5 File write function listing . . . . .	30
5-1 Transporting a file using GridOS . . . . .	34
5-2 Transporting a file using normal FTP client and server . . . . .	34
5-3 Reading and writing to a file . . . . .	35
6-1 GridOS vs Proftpd using standard ftp client . . . . .	37
6-2 GridOS ftp vs GridFTP using globus-url-copy as client . . . . .	37
6-3 GridOS ftp server/client vs GridFTP server/globus-url-copy . . . . .	38
7-1 ufl.edu logical structure . . . . .	40
7-2 ucsd.edu logical structure . . . . .	40
7-3 Find query result . . . . .	42
7-4 FSS architecture . . . . .	45

Abstract of Thesis Presented to the Graduate School  
of the University of Florida in Partial Fulfillment of the  
Requirements for the Degree of Master of Science

DESIGN AND IMPLEMENTATION OF GRIDOS: OPERATING SYSTEM  
SERVICES FOR GRID ARCHITECTURES

By

Pradeep Padala

December 2003

Chair: Joseph N. Wilson

Major Department: Computer and Information Science and Engineering

In this work, we demonstrate the power of providing a common set of operating system services to grid architectures, including high-performance I/O, communication, resource management and process management. A grid enables the sharing, selection, and aggregation of a wide variety of geographically distributed resources including supercomputers, storage systems, data sources, and specialized devices owned by different organizations administered with different policies. In the last few years, a number of exciting projects like Globus, Legion, and UNICORE developed the software infrastructure needed for grid computing. However, operating system support for grid computing is minimal or non-existent. Tool writers are forced to re-invent the wheel by implementing from scratch. This is error prone and often results in sub-optimal solutions. We have developed GridOS, a set of operating system services that facilitate grid computing. These services, developed as kernel modules make a normal commodity operating system like Linux highly suitable for grid computing. The modules are designed to be policy neutral, exploit commonality in various grid infrastructures and provide high-performance. Our

experiments with GridOS verify that there is dramatic improvement in performance when compared to the existing grid file transfer protocols like GridFTP. We have also developed a proof-of-concept middleware using GridOS.

## CHAPTER 1 INTRODUCTION

A grid[1] enables the sharing, selection, and aggregation of a wide variety of geographically distributed resources including supercomputers, storage systems, data sources and specialized devices owned by different organizations administered with different policies. Grids are typically used for solving large-scale resource and computing intensive problems in science, engineering, and commerce. In the last few years, a number of exciting projects like Globus[2], Legion[3] and UNICORE[4], developed the software infrastructure needed for grid computing. Various distributed computing problems have been solved using these tools and libraries. However, operating system support for grid computing is minimal or non-existent. Though these tools have been developed with different goals, they use a common set of services provided by the existing operating system to achieve different abstractions.

GridOS provides operating system services that support grid computing. It makes writing middleware easier and provides services that make a normal commodity operating system like Linux more suitable for grid computing. The services are designed as a set of kernel modules that can be inserted and removed with ease. The modules provide mechanisms for high performance I/O (`gridos_io`), communication (`gridos_comm`), resource management (`gridos_rm`), and process management (`gridos_pm`). These modules are designed to be policy neutral, easy to use, consistent and clean.

The goal of this body of work is to demonstrate the power of operating system services specific to grid computing. Well designed minimal, policy-neutral operating

system services can make a commodity linux operating system highly suitable for grid computing.

In chapter 2, we describe the motivation behind this work. Details of previous work and an overview of current trends are provided.

A clean design plays an important role in adding flexible and powerful services to an existing operating system. Chapter 3 gives a detailed describing of our design process. Our design principles and motivation behind choosing a layered architecture are explained. GridOS consists of four core modules including high performance i/o, resource management, process management and communication. Design considerations for each module are detailed.

Chapter 4 provides details of implementation. We developed the modules as kernel modules using Linux 2.4.19 kernel. Linux kernel module architecture is exploited to provide easy loading and unloading of the module. Specific details on implementing the ftp modules with details on module activation and de-activation are provided. We also developed middleware to ease the process of writing an application that uses GridOS services.

Chapter 5 shows various scenarios of GridOS usage. The chapter gives a pictorial view of GridOS operations.

Our experiments indicate that GridOS outperforms the standard ftp and GridFTP. Details about the experiments and graphs showing performance comparison of GridOS with basic FTP and GridFTP are in chapter 6.

We conclude with possible ideas for future work and improvements that can be done to GridOS.

## CHAPTER 2 MOTIVATION

This chapter presents the motivation behind this thesis. It provides an overview of grid computing and current trends in grid computing. The most prominent grid middleware Globus and Condor are explained briefly.

### 2.1 The Grid

Distributed computing has fascinated researchers all over the world for past two decades. Traditionally the focus has been on developing a unified homogeneous distributed system. Recently, there has been tremendous growth in wide area distributed computing leading to grid computing[1].

The term Grid is chosen as an analogy to electric power grid that provides consistent, pervasive, dependable, transparent access to electricity, irrespective of type and location of source. The primary focus in Grid computing is to harness the power of geographically distant supercomputers, and convert them into a big computing resource.

The Grid enables sharing, selection and aggregation of various resources including raw cpu cycles, storage systems, data sources and special services like application servers, etc. These resources may be geographically dispersed, operated by different organizations with different policies running on completely different operating systems.

Due to the recent explosion of Grid Technologies, there has been some confusion over the exact nature of a Grid. Dr. Foster provides a three point checklist[5] for evaluating grid systems.

A grid is a system that

- coordinates resources that are not subject to centralized control - A Grid integrates and coordinates resources and users that live within different control domains for example, the user's desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership, and so forth that arise in these settings. Otherwise, we are dealing with a local management system.
- uses standard, open, general-purpose protocols and interfaces - A Grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery, and resource access. It is important that these protocols and interfaces be standard and open. Otherwise, we are dealing with an application-specific system.
- delivers nontrivial qualities of service - A Grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts.

Grids are composed of VO(*Virtual Organization*)s, a conglomeration of network resources consisting of servers, desktop PCs, mainframes, clusters, etc. These VOs are managed by different organizations and may have different policies and security mechanisms. Grid enables sharing of resources between these heterogeneous VOs with a common set of open protocols.

There are enormous opportunities for application writers to exploit the large amount of computational and storage resource provided by the grid.

## 2.2 Grid Applications

Grid applications typically involve large-scale scientific computing that requires huge amount of cpu and data processing resources. The Grid Computing paradigm allows these huge applications to run smoothly. There are three classes of applications: compute intensive, data intensive and mixed applications.

Compute intensive applications such as finite element analysis, computational fluid dynamics, and design-of-experiment (DOE) simulations often require a large

amount of CPU power. These applications can easily be parallelized and can be run on the grid.

Data intensive applications require large inputs and produce huge amounts of data (sometimes on the order of petabytes). Various scientific applications, including High Energy Physics(HEP), Biology and Medical and Earth Observations(EO) applications, analyze and derive large amounts of data. There is also a growing need to share this data among various scientific communities. The communities of researchers are often large and geographically distributed, as are the computing and storage resources used to analyze the data. The *data grid*[6] provides a way of managing this distributed data in a seamless way.

Some applications are both compute and data intensive.

### 2.3 Grid Middleware

To achieve the goals of the Grid, existing technologies are extended and various new technologies are developed. These technologies provide a framework to do various Grid operations with minimum interaction with underlying platform, architecture and operating system.

Grid middleware provides facilities to use the grid for applications and users. Middleware like Globus[2], Legion[3] and UNICORE[4] provide software infrastructure to tackle various challenges of computational and data grids. *Data middleware*, which usually is part of general purpose grid middleware, provides facilities for data management.

Various middleware products have been developed in recent years that are widely in various scenarios. In their earlier versions, grid middleware provided incompatible methods to access, manipulate and store the data. The Open Grid Services Architecture(OGSA)[7] defines the standard communication protocols and formats for building truly interoperable grid systems. The Open Grid Services Infrastructure(OGSI)[8] provides the specification for building various grid services

envisioned in OGSA. These specifications have standardized grid services describing how they operate and interact. OGSi provides detailed WSDL specifications for standard interfaces that define a grid service.

The third version of the Globus toolkit (GT3) provides implementation of OGSi specification and provides various OGSi-compliant basic services including managed job service, index service and reliable file transfer service (RFT). GT3 also provides higher level data services including RLS (Replica Location Service). The data management services RFT and RLS provide the basic mechanisms for data management.

In the following sections, we explore popular grid middleware products and their architectures.

## 2.4 Globus

Globus[2] provides a software infrastructure that enables applications to handle distributed heterogeneous computing resources as a single virtual machine. The Globus project is a U.S. multi-institutional research effort that seeks to enable the construction of computational Grids. A computational Grid, in this context, is a hardware and software infrastructure that provides dependable, consistent, and pervasive access to highend computational capabilities, despite the geographical distribution of both resources and users. Globus provides basic services and capabilities that are required to construct a computational Grid. The toolkit consists of a set of components that implement basic services, such as security, resource location, resource management, and communications.

It is necessary for computational Grids to support a wide variety of applications and programming paradigms. Consequently, rather than providing a uniform programming model, such as the object-oriented model or message-passing model, the Globus provides a bag of services from which developers of specific tools or applications can use to meet their own particular needs. This methodology is only

possible when the services are distinct and have well-defined interfaces (APIs) that can be incorporated into applications or tools in an incremental fashion.

Globus is constructed as a layered architecture in which high-level global services are built upon essential low-level, core local services. The Globus toolkit is modular and an application can exploit Globus features, such as resource management or information infrastructure without using Globus communication libraries.

Before we try to understand the Globus architecture, let us have a quick look at the layered grid architecture. Figure 2–1 shows the layered structure of the grid architecture.

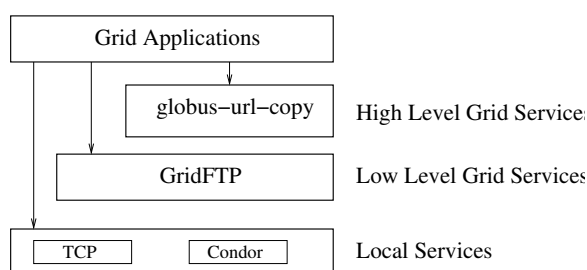


Figure 2–1: Layered grid architecture

As we can see, the applications can use high-level tools like `globus-url-copy` or use the low level libraries like `globus_ftp_client`.

The Globus toolkit is divided into three pillars. Each pillar represents a primary component that addresses various aspects of grid computing.

All the components use GSI (Grid Security Infrastructure)[9] for security at the connection layer. Each of these components has a primary protocol providing the basic functionality. GRAM (Globus Resource Allocation Manager)[10] implements a resource management protocol, MDS (Metacomputing Directory Service)[11] implements an information services protocol, and GridFTP[12] implements a data transfer protocol.

### 2.4.1 Security

GSI provides a security framework for Globus. All other components use this layer to connect to the world. It provides various features which are important and necessary in a grid environment

- *Secure authentication and authorization*: GSI provides features to securely authenticate a user and authorize access to resources
- *Single sign-on*: One of the problems associated with distributed systems is the task of authenticating a user on each system that he/she wants to execute a task. GSI provides single sign-on using which a user has to authenticate only once and his proxy can execute tasks in various administrative domains. The proxies have a limited lifetime to avoid compromise of credentials.
- *Interoperability with local security solutions*: Usually VOs have different local security policies within their domain. GSI provides mechanisms to translate between inter domain security operations and local mechanisms.
- *Uniform credentials/certification*: GSI uses X.509 certificate data format and public key encryption for user credentials. Users have to obtain a certificate to access Globus resources.

### 2.4.2 Data Management

In scientific experiments, efficient data management plays an important role. These applications access and analyze large amounts of data (sometimes of the order of peta bytes) and require high-performance and reliable data transfers. Globus provides GridFTP, a high-performance, secure, and reliable data transfer protocol optimized for high-bandwidth wide-area networks. GridFTP extends the popular FTP with features like multiple TCP streams, striped data transfers, etc. Some of the features are

- *Secure data transfer*: GridFTP uses GSI to provide secure data transfer mechanisms
- *Parallel data transfer*: Multiple TCP streams are used to efficiently utilize the high bandwidth of wide-area networks.
- *Partial data transfer*
- *Third party data transfer*: Data can be transferred from a server to another server using third party control.

- *Striped data transfer*: This is an important feature which allows data to be partitioned across multiple servers. As a result, with efficient data partitions, aggregate bandwidth can be greatly improved.

### 2.4.3 Resource Management

Resource management is an important component of a grid. Globus provides features to allocate and use resources. Resource management component consists of three main components

- *RSL (Resource Specification Language)*: RSL provides a common language used by all the components in resource management architecture to perform their functions. A user uses RSL to specify resource requirements, e.g. `executable = globus-url-copy`
- *GRAM (Globus Resource Allocation Manager)*: GRAM manages allocation of resources for execution of jobs. It provides a standard interface to local resource management tools like LSF, Condor etc. Users can query GRAM to get the status of their submitted jobs.
- *DUROC (Dynamically Updated Request Online Co-allocator)*: Co-allocation is the process of allocating multiple resources at various sites required by grid applications. This is an important aspect of grid computation. DUROC provides an API to allocate resources in multiple sites.

### 2.4.4 Information Services

The Globus information service is called Metacomputing Directory Service(MDS). It provides means of discovering, characterizing and monitoring resources and services in a heterogeneous environment. MDS uses LDAP for providing uniform access. MDS has two main components

- *GRIS (Grid Resource Information Service)*: GRIS provides uniform means of querying resources for their current configuration, capabilities, and status. This is used by GRAM to provide status of jobs to the user. *GIIS (Grid Index Information Service)*: GIIS can be used to discover resources matching a criterion. It provides a coherent system image of the grid that can be explored by applications.

Globus can be thought of as a Grid computing framework based on a set of APIs to the underlying services. Globus provides application developers with various means of implementing a range of services to provide a wide-area application execution environment.

## 2.5 Condor

The Condor[13] project started as providing a framework for high throughput computing. It provides a framework for workload management in a pool of workstations. Condor allows effective utilization of available cpu cycles by detecting idle workstations. Condor provides a job queueing mechanism, scheduling policies, priorities, resource monitoring and resource management. It is often used as local scheduler within a virtual organization. In this section, we briefly look at various features provided by Condor.

Some of the features provides by Condor include

- *Framework for Opportunistic Computing*: Opportunistic Computing is the utilization of all the available resource without requiring 100% availability. Condor effectively harnesses the available CPU cycles by sending jobs to idle workstations. It provides queuing, scheduling and planning facilities to achieve this.
- *ClassAds*: ClassAds[14] provide a mechanisms for match-making. Both Buyers and Sellers of resources publish a *classified advertisement* and Condor's MatchMaker matches the requests with available resources. The MatchMaker employs a simple matching algorithm that follows the constraints and policies specified in the advertisements. A ClassAd example is shown in figure 2-2.
- *Checkpointing and Migration*: When users start using idle workstations, sometimes it is necessary to move the running jobs to other available idle workstations. This requires checkpointing and migration facilities. Condor provides these features with some constraints. When a job needs to be migrated, Condor creates a snapshot of the job called a checkpoint, which includes detailed account of process state. When the job is re-started on another machine, the checkpoint file is used to re-create its state. Programs have to be re-linked with the Condor checkpointing libraries to exploit this feature.

```
[  
  
    Type = "Job";  
  
    Owner = "ppadala";  
  
    Universe = "Standard";  
  
    Requirements = (other.OpSys == "Redhat Linux" && other.DiskSpace > 500M);  
  
    ClusterID = 1234;  
  
    JobID = 0;  
  
    Env = ""  
  
    Groups = {"condor", "griphyn"};  
  
    Files = { [ name = "simulation_data.txt", type = "input"],  
             [ name = "analysis.dat", type = "output" ] }  
  
    ....  
]
```

Figure 2-2: A ClassAd example

- *Split Execution System*: Condor provides interposition agents so that a program executing on a remote system can transparently make a system call to the originating system. This is useful in running legacy applications unmodified on a remote system.
- *Condor-G: A Computation Management Agent for Grid Computing*: Condor-G provides facilities to combine Condor and Globus technologies. Condor-G can be used both as a frontend and backend to Globus toolkit mechanisms. It provides a consistent interface to submit jobs to resources managed in different ways.

## 2.6 Legion

Legion[3] is an object-based meta computing system that envisions providing a virtual world-wide computer view to the user. Using Legion, users see a single computer with enormous resources. Groups of users can construct a virtual workspace to collaborate and access resources. Legion sits on top of user's operating system and using its scheduling and resource management algorithms provides this view.

Legion provides various features including

- *Object-based Meta System*: Legion provides an object-based meta system in which core objects like host objects and vault(storage) objects. The core objects provide the basic framework for building advanced services. Every thing in Legion is an object and the objects are persistent. Users can create new classes that allows a powerful mechanism to extend Legion capabilities.
- *Legion File System*: Legion file system is developed to cope with the diverse requirements of wide-area collaborations. It provides a fully integrated architecture that provides location independent naming, security, scalability, extensibility and adaptability.
- *Resource Management Services*: Legion provides an object-based resource management system that allows user-defined schedulers. The resource model contains basic resources (hosts and vaults), the information database (the Collection), the scheduler implementor (the Enactor) and an execution monitor. See [15] for more details on these components.

## 2.7 Previous Work

The primary motivation for this work comes from chapter twenty of the book *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, edited by Ian Foster et al[1], in which the authors discuss the challenges in the operating system

interfaces for grid architectures. The book discusses various principles but stops short of implementation details.

While there has been little work on Grid Operating System interfaces, there has been tremendous development in grid middleware. Projects like Globus and Legion provide elaborate software infrastructure for writing grid applications. These tools and libraries have to cope with the existing operating system services that are not designed for high-performance computing. As a result, they are forced to implement some commonly used high-performance optimizations like multiple TCP streams and TCP buffer size negotiation that more suitably should be implemented in the operating system's kernel. These tools, though quite different, often use the same set of low-level services like resource management, process management, and high-performance I/O. For example, both Globus and Legion have to query the operating system for resources and maintain information for effective resource management. As there is no operating system support for these low-level services, middleware developers must implement them from scratch. This is error prone and often results in sub-optimal solutions.

There have been some attempts to add operating system services that perform high performance computing. WebOS[16] provides operating system services for wide area applications provides services for wide area applications. PODOS[17], a performance oriented distributed operating system, is similar and provides high performance through optimized services in the kernel. This work succeeds in providing high performance. But due to the extensive changes made to the kernel, it is difficult to port this work to newer kernels. It is also difficult to extend the services due to its monolithic structure. GridOS addresses these problems by providing a modular architecture that requires minimal changes to the kernel and makes it easy to extend the services.

Apart from the above mentioned work, there has been great amount of research done in distributed operating systems. Systems like Amoeba[18] and Sprite[19] had great impact on the distributed programming paradigm. We have incorporated some of the principles used in designing these distributed operating systems in GridOS.

## CHAPTER 3 GRIDOS DESIGN

### 3.1 Core Principles

The following principles drive the design of GridOS. These principles derive from the fact that the toolkits like Globus require a common set of services from the underlying operating system.

- *Modularity.* The key principle in GridOS is to provide modularity. The components of GridOS should be modular and interact with well-defined interfaces. The Linux kernel module architecture is exploited to provide a clean modular functionality.
- *Policy Neutrality.* GridOS follows one of the guiding principles of design of operating systems: policy-free mechanisms. Instead of providing a *black box* implementation that takes care of all possibilities, the modules provide a policy-free API that can be used to develop high level services like GridFTP. Higher level middleware, having more information about the application should be able to setup policies using GridOS APIs.
- *Universality of Infrastructure.* GridOS provides a basic set of services that are common to prevalent grid software infrastructures. It should be easy to build radically different toolkits like Globus (a set of independent services) and Legion (an object-based meta-systems infrastructure).
- *Minimal Core Operating System Changes.* We do not want to make extensive modifications to the core operating system as that would make it difficult to keep up with new OS versions.

These guiding principles led us to develop a layered architecture (Figure 3-1). The lowest layer contains the primary GridOS modules that provide high-performance grid computing facilities. These modules are orthogonal and provide basic mechanisms. They do not mandate any policies. The upper layers provide specific services similar to GridFTP, GASS[20].

This approach is motivated by the observation that the toolkits usually make policy decisions on behalf of the grid applications. The toolkit, knowing the

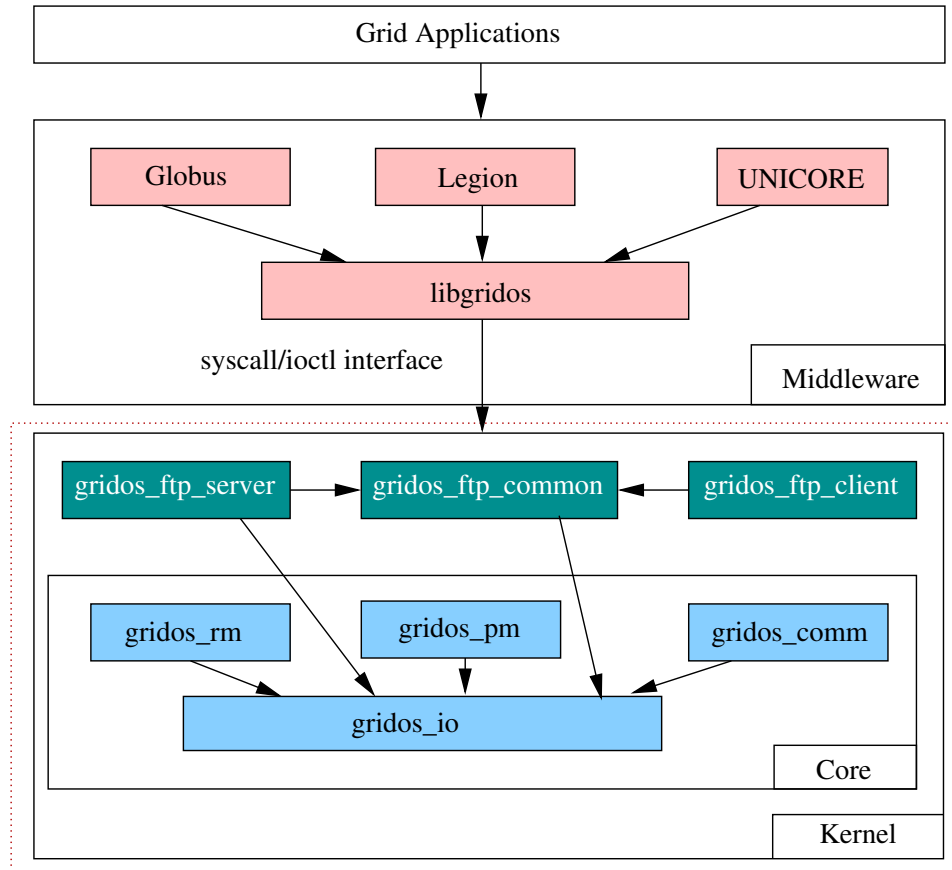


Figure 3–1: Major modules and structure of GridOS

application requirements and having domain knowledge, can make a better decision about what policy to employ. This also encourages a wide variety of toolkits to be developed on top of GridOS.

The GridOS core services are provided by an `ioctl` and `system` call interfaces. A library called `libgridos` is developed to provide wrappers to the low-level interfaces. Various kinds of middleware can be developed on top of the library. Toolkit developers should normally be able to use the `libgridos` API for accessing GridOS services.

### 3.2 Core Modules

In the following sections we describe GridOS modules. We explain the implementation details in a later chapter.

### 3.2.1 gridos\_io: High-Performance I/O Module

High performance I/O is an important aspect of grid applications. This module provides High-Performance network I/O capabilities for GridOS. In an increasing number of scientific disciplines, large amounts of data (sometimes on the order of petabytes) are accessed and analyzed regularly. For transporting these large amounts of data, high-speed WANs are used. These networks have high bandwidth and large round trip times (RTT), sometimes referred to as “Long Fat Networks” (LFNs, pronounced “elefan(t)s”) [21].

In order to take full advantage of these high speed networks, various operating system parameters must be tuned and optimized. Several techniques for achieving high-performance are outlined below. It is important to realize that some of these techniques are not additive, but rather complementary. Applying a single technique may have little or no effect, because the absence of any one of the techniques can create a bottleneck.

No User-Space Copying. A user-space FTP server requires the kernel to copy the network buffers to user-space buffers to retrieve data from the client. It then requires the kernel to copy the user-space buffer to file system buffers to write data to a file. This incurs a large overhead due to time spent in copying.

Because `gridos_io` and `gridos_ftp` are kernel modules that handle both network and file system I/O, thus double copying can be avoided.

TCP WAN Throughput. TCP (Transmissions Control Protocol)[22] is the standard transport layer used over IP networks. TCP uses the congestion window (CWND), to determine how many packets can be sent before waiting for an acknowledgment. On wide area networks, if CWND is too small, long network delays will cause decreased throughput. This follows directly from Little’s Law[23] that

$$WindowSize = Bandwidth * Delay$$

The bandwidth is the bandwidth of the slowest link in the path. This can be measured with tools like `pipechar` and `pchar`. The delay can be measured by calculating the round trip time (RTT) using `ping` or `traceroute`.

The TCP *slow start* and *congestion avoidance* algorithms determine the size of the congestion window. The kernel buffers are allocated depending on the maximum size of the congestion window.

To get maximal throughput it is essential to use optimal TCP send and receive kernel buffer sizes for the WAN link we are using. If the buffers are too small, the TCP congestion window will never fully open up. If the buffers are too large, a fast sender can overrun a slow receiver, and the TCP window will shut down. The optimal buffer size for a link is  $\text{bandwidth} * \text{RTT}$ [24, 25]

Apart from the **default** send and receive buffer sizes, **maximum** send and receive buffer sizes must be tuned. The default maximum buffer size for Linux is only 64KB which is quite low for high-speed WANs.

The `gridos_io` module provides various ways of controlling these buffer sizes. Using the `gridos` system call, default values for the buffers can be specified. This may be done by the system administrator who has knowledge about the networks. `gridos_io` measures the throughput from time to time and updates the buffer sizes accordingly.

### 3.2.2 gridos\_comm: Communication Module

Communication is an important aspect of a distributed application. Often applications require various modes of communication depending on the underlying medium, destination and target application. For example, an application transferring a huge file requires the network pipe to be full and wants to utilize all the available bandwidth. On the other hand, an application using MPI calls to communicate with jobs distributed over a wide area network, requires a quick delivery of short messages.

The Foster et al. summarizes the requirements for multiple methods of communication in [26]. Our interpretation of these requirements in the context of GridOS is as follows.

- *Separate specification of communication interface and communication method:* Various communication methods are used to send and receive messages. For example, TCP/IP networks provide reliable (TCP) and un-reliable (UDP) communication mechanisms. But, the interface to these low-level communication primitives need to be consistent and should be separate from the underlying mechanisms. Globus and Legion toolkits already achieve this but with considerable difficulty. GridOS offers consistent primitives to asynchronous, synchronous and reliable, unreliable communications.
- *Automatic selection:* Automatic selection of communication heuristics is required for ease of programming. Though this is in the domain of middleware, GridOS tries to select appropriate communication mechanism using simple heuristics. Note that all the parameters of GridOS can be manually selected by the middleware layer as explained below.
- *Manual selection:* Grid middleware should allow the user to manually select communication methods. The middleware requires manual selection primitives from underlying system like GridOS.
- *Error Handling & Feedback:* Method specific feedback is required in some applications. For example, a multimedia application might want to know about the jitter and frame loss rate so that it can optimize the communication. GridOS provides this information to the middleware through well-defined interfaces.

The communication module is designed so that middleware writers can provide multiple communication methods that support both automatic and programmer-assisted method selection. Grid applications are heterogeneous not only in their computational requirements, but also in their types of communication. Different communication methods differ in usage of network interfaces, low-level protocols and data encodings and may have different quality of service requirements. This module allows communication operations to be specified independent of methods used to implement them.

The module also provides multi-threaded communication which is used in implementing the FTP module. Using the library, various high-level mechanisms like MPI (Message Passing Interface)[27] can be easily implemented.

### 3.2.3 gridos\_rm: Resource Management Module

Grid systems allow applications to assemble and use collections of resources on an as-needed basis, without regard to physical location. Grid middleware and other software architecture that manage resources have to locate and allocate resources according to application requirements. They also have to manage other activities like authentication and process creation that are required to prepare a resource to use. See [10] for details on various issues involved in resource management.

Interestingly, these software infrastructure use a common set of services. Gridos\_rm provides these facilities so that software infrastructure can be developed to address higher-level issues like co-allocation, online-control etc.

The module provides facilities to query and use resources. The module maintains information about current resource usage of processes started by local operating system and GridOS modules. It also provides facilities to reserve resources. These facilities can be used to develop systems like Condor[13], which discovers idle resources on a network.

### 3.2.4 gridos\_pm: Process Management Module

This module allows creation and management of processes in GridOS. It provides a global PID (GPID) for every process in GridOS and provides communication primitives which can be used on top of `gridos_comm` for processes to communicate among themselves[28].

This feature allows disparate toolkits like Condor and Globus to use a common mechanism for creating processes and pass process ids. The global PID is created by combining the host identifier and local process id. The module also provides

services for process accounting. This feature is important in accounting for the jobs which are transported to GridOS.

### 3.3 Additional Modules

#### 3.3.1 gridos\_ftp\_common

This module provides facilities common to any FTP client and server. This includes parsing of FTP commands, handling of FTP responses etc.

#### 3.3.2 gridos\_ftp\_server

This module implements a simple FTP server in kernel space. Due to its design, copying of buffers between user and kernel space is avoided. The server does not start a new process for each client as is usually done in typical FTP servers. This incurs low overhead and yields high-performance. The server also uses `gridos_io` facilities to monitor bandwidth and adjust the file system buffer sizes. The file system buffers are changed depending on the file size to get maximum overlap between network and disk I/O.

The module is designed with security in mind. Even while operating in the kernel mode it drops all privileges and switches to an unprivileged user-id and group-id. It also chroots to FTP top level directory `docroot` which can be configured dynamically.

#### 3.3.3 gridos\_ftp\_client

This module implements a simple FTP client in the kernel. The main purpose of this module is to decrease the overhead of writing or reading file on the client side. Our experiments indicate that primary overhead on the client side is the time spent in reading and writing files. By carefully optimizing the file system buffers to achieve maximum overlap between network and disk I/O, high-performance is achieved.

## CHAPTER 4 IMPLEMENTATION

We have implemented a subset of GridOS on Linux using the stable 2.4.19 kernel. The Linux kernel module architecture is exploited to provide ease of installation for users and ease of modification for developers. The code is divided into a small patch to the core kernel and a set of kernel modules. The patch is designed to be minimal and adds the basic `ioctl` and `syscall` interfaces to GridOS.

The modules are inserted and removed using `insmod` and `rmmmod` utilities. The dependencies between the modules are resolved using `modules.dep`. As a result, if `gridos_comm` is inserted without inserting `gridos_io` the kernel would find that there is a dependency and insert `gridos_io` automatically.

Currently, we have implemented `gridos_io`, `gridos_ftp_server`, `gridos_ftp_client` and the library wrapper `libgridos`. We have also implemented a simple middleware called `gridos-middleware` as a proof-of-concept showing the ease with which middleware can be developed. The following sections explain the internals of the modules.

### 4.1 Linux Kernel Module Architecture

The Linux kernel module architecture provides micro kernel facilities to a monolithic design. In the olden days, to add features to the Linux kernel, one had to modify the source and build a new kernel. This is still true for core parts of the kernel like the scheduler. Loadable kernel modules are developed as a way of adding non-core features easily into the kernel. Once loaded into memory, the kernel module becomes part of the running kernel.

There are many advantages to using loadable kernel modules. Some of the advantages are

- No need to re-build the kernel. As a result, modules can be added to and removed from a running kernel without rebooting the machine or compiling a new kernel.
- Easy to diagnose problems. If a loaded driver is causing problems in a perfectly running kernel, the problem is probably with the driver.
- Can save memory. Administrators can unload modules that are not used and free up memory. Linux provides automatic module loading and unloading as well.
- Provides modularity. Different people can work on various modules without worrying about making conflicting changes to the kernel.

#### 4.2 Kernel Level Web/Ftp Server (tux)

Tux is an HTTP protocol layer and a web server object cache integrated into the Linux Kernel. It can be thought of as a kernel-level web server. TUX stands for Threaded linUX http layer.

Tux is an interesting concept similar to the methods used in High-Throughput Computing. It provides high-performance web server by avoiding copying of network buffers between user and kernel mode. Usually web servers are big piece of software written in user-mode. When the data arrives over network, kernel copies the data into user-mode buffers from kernel network buffers. When high-performance is needed, this becomes a bottleneck and may produce lot of overhead. If the web server is just serving static content, then this copying of data back and forth from user to kernel mode can be avoided by having a small kernel web server. Tux fills this gap perfectly by serving static content directly from kernel mode and leaving dynamic content like CGI to an external web server like Apache.

This section provides a brief overview and code review of Tux. I also indicate various parts of tux that are used in implementing GridOS modules.

### 4.2.1 User Level Access to kernel HTTP layer

User processes can access the Tux server by using the `sys_tux` system call.

Using this system call, user processes (usually the rc scripts) can drive the server.

The caller specifies various actions like startup, shutdown server, start thread etc.

- *Startup*: The startup action sets up TUX data structures depending on the parameters passed from user mode. These include the number of threads, buffer sizes, etc.
- *Shutdown*: This action shuts down TUX cleaning up the memory used by TUX.
- *Register Module*: TUX provides module facilities so that a user can write a user-mode module to handle specific activity like CGI query handling. The register action registers a handler that will be called based on a specific action.
- *Un-register Module*: This action unregisters the module.
- *Start Threads*: This is one of the important actions of TUX. TUX does its own handling of kernel threads. As a result, TUX is much faster than other similar software using kernel threads.

Figure 4-1 shows relevant code. Note that this code and other code snippets are simplified for clarity.

### 4.2.2 Listening & Accepting Messages

The listener threads in tux listens to any requests and accepts the connections. The function `start_listening` listens on the specified socket and accepts connections through `accept_requests` function. The `start_listening` function is used with small modifications in GridOS to accept incoming connections. Figure 4-2 shows relevant pieces of code.

An event loop waits for accepted connections. The event loop goes through all the threads checking for pending connections requiring processing. Once an accepted connection is found, it removes it from the accept queue and processes the content of request. Figure 4-3 shows the event loop.

The `proto->got_request` is the function `http_got_request` which calls `parse_request`. `parse_request` calls `parse_message` which will parse the message and finally calls

```

switch (action) {
    case TUX_ACTION_STARTUP:
        lock_kernel();
        ret = user_req_startup();
        unlock_kernel();
        goto out;

    case TUX_ACTION_SHUTDOWN:
        /* code similar to above */

    case TUX_ACTION_REGISTER_MODULE:
        ret = user_register_module(u_info);
        goto out;

    case TUX_ACTION_UNREGISTER_MODULE:
        ret = user_unregister_module(u_info);
        goto out;

    case TUX_ACTION_STARTTHREAD:
    {
        unsigned int nr;

        /* get the number of threads */
        ret = copy_from_user(&nr, &u_info->thread_nr,
                            sizeof(int));
        error_handling;

        ti = threadinfo + nr;

        /* stuff added to process structure */
        current->tux_info = ti;
        current->tux_exit = tux_exit;

        lock_kernel();
        /* the real thing */
        ret = user_req_start_thread(ti);
        unlock_kernel();

        /* after returning set tux_info and tux_exit to null */
    }
}

```

Figure 4-1: Tux actions

```

struct socket * start_listening(tux_socket_t *listen, int nr)
{
    err = sock_create(PF_INET, SOCK_STREAM, IPPROTO_TCP, &sock);
    if (err < 0) {
        goto err;
    }

    /* Bind the socket: */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(addr);
    sin.sin_port = htons(port);

    sk = sock->sk;
    sk->reuse = 1;
    sk->urinline = 1;

#define IP(n) ((unsigned char *)&addr)[n]
    err = sock->ops->bind(sock, (struct sockaddr*)&sin, sizeof(sin));
    if (err < 0) {
        goto err;
    }

    /* sk structure contents are filled with appropriate values
       for performance gain. */
    err = sock->ops->listen(sock, tux_max_backlog);
    if (err) {
        goto err;
    }
    return sock;

    error_handling;
}

```

Figure 4-2: Connection accept

```

switch (action) {

    case TUX_ACTION_EVENTLOOP:
eventloop:
        req = ti->userspace_req;
        if (req)
            zap_userspace_req(req);
        ret = event_loop(ti);
        goto out_userreq;

        .
        .
        .
}
static int event_loop (threadinfo_t *ti)
{
    .
    .
    __set_task_state(current, TASK_INTERRUPTIBLE);
    work_done = 0;
    if (accept_pending(ti))
        work_done = accept_requests(ti);
    .
    .
}

```

Figure 4-3: Event loop

```

/*
 * Puts newly accepted connections into the inputqueue. This is the
 * first step in the life of a TUX request.
 */
int accept_requests (threadinfo_t *ti)
{
    .
    .
    .
    tp1 = &sock->sk->tp_pinfo.af_tcp;
    /*
     * Quick test to see if there are connections on the queue.
     * This is cheaper than accept() itself because this saves us
     * the allocation of a new socket. (Which doesn't seem to be
     * used anyway)
     */
    if (tp1->accept_queue) {
        tux_proto_t *proto;

        /* set task state to running */

        new_sock = sock_alloc();
        error_handling;

        new_sock->type = sock->type;
        new_sock->ops = sock->ops;

        error = sock->ops->accept(sock, new_sock, 0_NONBLOCK);
        .
        .
        .
        proto = req->proto = tux_listen->proto;
        proto->got_request(req); /* this will call http_got_request */
    }
}

```

Figure 4-4: Accept request

`http_parse_message`.

The request functions have been modified in GridOS to handle GridOS specific connections. Other code in GridOS is written from scratch.

## 4.3 Modules

### 4.3.1 Activation

The modules activation and de-activation is done with usual `insmod` and `rmmod` commands. Apart from the basic module dependency features provided by the Linux kernel, I developed a mechanism to check whether a required GridOS is moduled is loaded. Each GridOS module after startup updates a global kernel structure (after taking care of the synchronization issues). When a GridOS module starts up, it checks whether all core modules are present or not. Then, it checks for the existence of required additional modules.

### 4.3.2 IO Module

The globus IO module implementation is divided into two APIs, one each for the network and the file system. The IO module is designed to minimize copy operations and let the data flow remain within the kernel.

The network API includes functions to read and write data from a gridos managed socket. Both blocking and non-blocking read calls are provided. Gridos FTP modules make extensive use of non-blocking reads for high-performance. It also has functions to set various buffer management options.

- *gridos\_io\_sync\_read*: This function is used to read data from a gridos managed socket in blocking mode.
- *gridos\_io\_async\_read*: This function is used to read data from a gridos managed socket in non-blocking mode
- *gridos\_io\_write*: This function writes data to the gridos managed socket
- *gridos\_io\_buffer\_setopt*: This function sets options for buffer management. The options include setting of TCP send and receive buffer sizes, maximum TCP buffer size etc.
- *gridos\_io\_buffer\_getopt*: This function returns the current buffer management options.

```

int gridos_io_file_write(const char *buf, const char *dst, int size)
{
    struct file *f = NULL;
    int flags, len;
    mm_segment_t oldmm;
    int mode = 0600;

    flags = O_WRONLY;
    if(!dst) {
        printk(KERN_ERR "Destination file name is NULL\n");
        return -1;
    }
    f = filp_open(dst, flags, mode);
    if (!f || !f->f_op || !f->f_op->write) {
        printk(KERN_ERR "File (write) object is NULL \n");
        return -1;
    }
    f->f_pos = 0;

    oldmm = get_fs(); set_fs(KERNEL_DS);
    len = f->f_op->write(f, buf, size, &f->f_pos);
    set_fs(oldmm);
    if (f->f_op && f->f_op->flush) {
        lock_kernel();
        f->f_op->flush(f);
        unlock_kernel();
    }
    fput(f);
    printk(KERN_INFO "Wrote %d bytes\n", len);
    return len;
}

```

Figure 4–5: File write function listing

The file system API has similar functions for reading and writing data to files. These functions call the Linux Kernel’s VFS (Virtual File System) functions to achieve this. Here is some sample code showing implementation details (simplified for clarity)

The file system API also has a higher-level function `gridos_file_copy` which can be used to copy a file locally. This can be used for high-performance local copying of files.

`gridos_io` also has facilities to compressed gzip data stream. This can be configured by setting the `compression` option.

### 4.3.3 FTP Client and Server Modules

The FTP server API allows the user to create an FTP server on a specified port. Various configuration options like `docroot` (top level directory for anonymous FTP) `threads` (number of simultaneous threads) etc can be set to control the behaviors of FTP module. Dynamic configuration can be done via Linux's `sysctl` mechanism. This provides access to FTP module features through `/proc` interface which can be used to read and modify various variables in the kernel address space.

Threading Model. For high performance kernel threads are used to satisfy simultaneous connections. The thread model has been adapted from TUX, a kernel web server. The model is similar to the FLASH web server[29] where requests are handled simultaneously while managing any disk I/O asynchronously in separate threads.

There are two thread pools. The first pool of threads is I/O or *cache-miss* thread pool. These threads populate the buffers asynchronously at the request of listener threads. The number of I/O threads can be changed by setting the `num_threads` configuration option. All the threads are started at the start of `gridos_ftp_server` module.

The second pool of threads is *fast* or *listener* threads. These threads handle requests and send responses to clients.

## 4.4 Library Wrapper

There are three ways of controlling `gridos` behavior from user-space.

- Through system call `sys_gridos`
- Using `ioctl` on `gridos` device
- Using `/proc` interface

libgridos provides an easy-to-use interface to these three low level interfaces.

Currently, C function wrappers are provided to

- Read and write from/to the network
- Read and write from/to files
- Set configuration options
- Start and Stop FTP server
- Use the FTP client

#### 4.5 GridOS Middleware

One of our design goals is to ease the development of middleware like Globus.

We have developed a small middleware prototype, which provides uniform access to files whether they are located remotely or locally. The operations are similar to the operations provided by Globus GASS. We developed this library in just a few days and it shows how easy it is to develop middleware using GridOS.

Various kinds of applications can be developed using the middleware.

## CHAPTER 5 SCENARIOS

### 5.1 Scenario #1: Transporting a file

In this section, we show module interaction while transporting a file. The file transporting is initiated by a user-space application like `gridos-middleware` that uses `sys_gridos`. Depending on the parameters various GridOS modules do the required work. In this case `gridos_ftp_client` calls `gridos_io_file_read` which reads the file into file system buffers. The buffers are used directly to initiate the network IO. On the server side the network buffers are handed over to `gridos_ftp_server` which can initiate a file writing with `gridos_io_file_write`. Figure 5–1 shows the scenario in detail. The process of starting the ftp server is not shown in the figure.

In comparison, figure 5–2 shows file transfer using a normal ftp client and server. In this scenario, you can see that double copying is done between user-mode and kernel-mode. In the GridOS scenario double copying is avoided by handling the transfers within the kernel.

### 5.2 Scenario #2: Reading and Writing to a file locally

In this scenario, we show the usage of `gridos_file_copy` that can be used to copy a file locally. As there is no copying between user and kernel mode, the performance is improved. Figure 5–3 shows the scenario in detail.

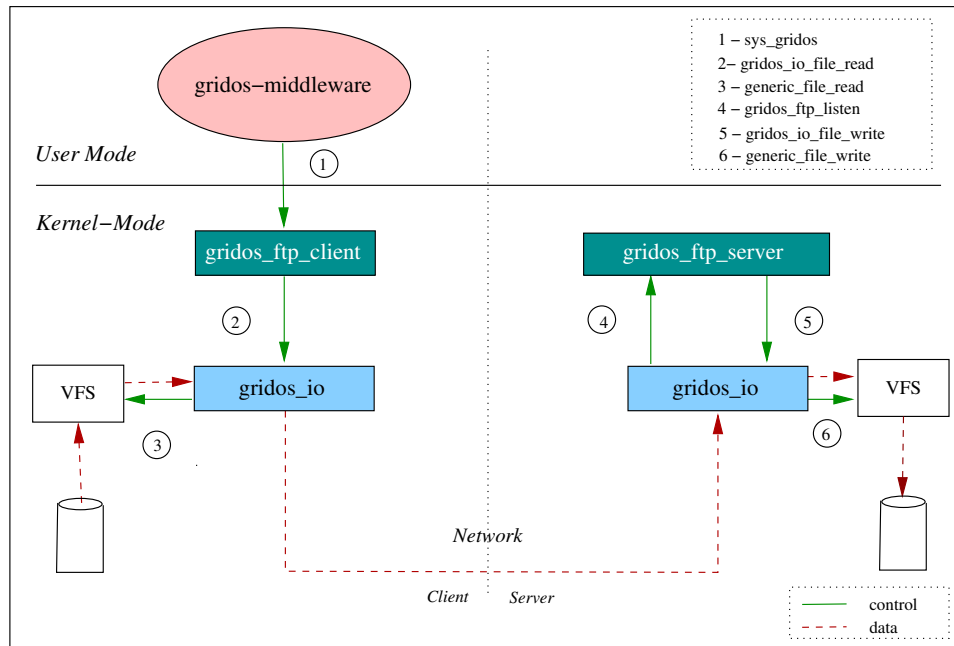


Figure 5-1: Transporting a file using GridOS

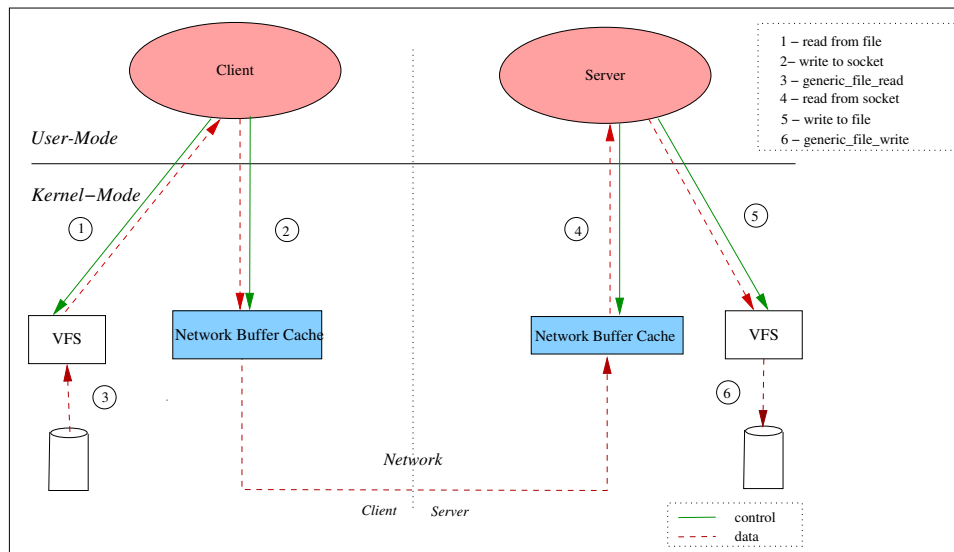


Figure 5-2: Transporting a file using normal FTP client and server

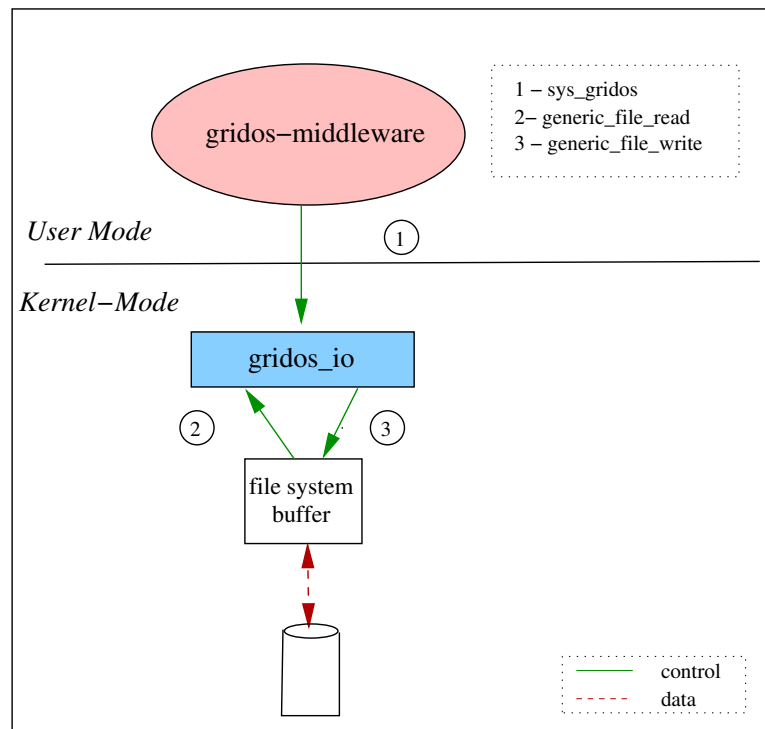


Figure 5-3: Reading and writing to a file

## CHAPTER 6 PERFORMANCE

We have conducted various experiments to measure the performance of GridOS comparing it to standard OS services and Globus services. The use of GridOS services results in a dramatic increase in throughput in our experiments. First, we compare ftp performance using GridOS and the standard proftpd shipped with Mandrake Linux, which showcases the advantages of zero-copy I/O. Then, we compare performance of transporting a file using GridFTP server and GridOS ftp using `globus-url-copy` as the client. This reveals an important bottleneck in high-performance I/O: file writing overhead. Finally, we compare performance of file transfer using GridOS ftp client and server and `globus-url-copy` and GridFTP server.

The experiments confirm that on average GridOS is 4 times faster than ftp and 1.5 times faster than GridFTP.

### 6.1 Test Environment and Methodology

The tests are conducted in an experimental grid setup in the self-managed network of CISE at UFL. Globus toolkit version 2.2 is used for setting up the grid. The two testing machines were connected over 100Mbps Ethernet network. The machines were kept idle while running the tests so as not to allow other processes running on these machines affect the performance numbers.

In each experiment files of sizes varying from 1KB to 512MB were used and the total time taken to transport each of these files was calculated. For each transfer the average of 5 runs for each file was taken. To avoid the affects of serving from the cache, the testing machine was rebooted after each run.

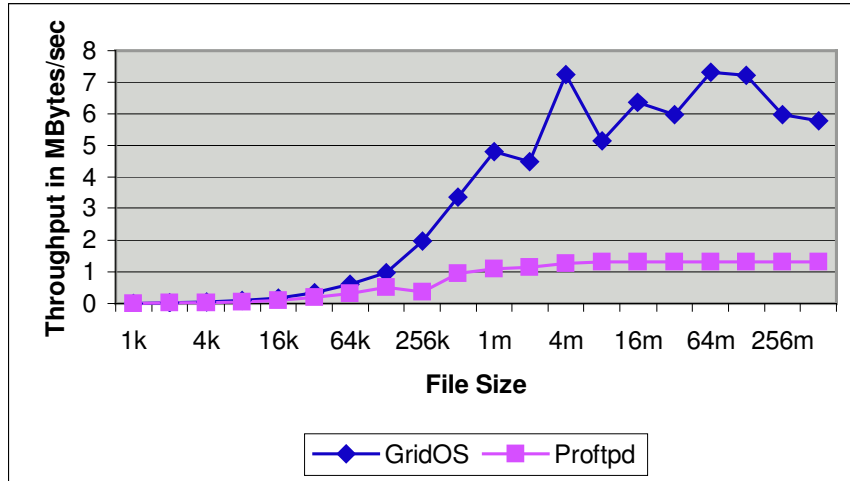


Figure 6–1: GridOS vs Proftpd using standard ftp client

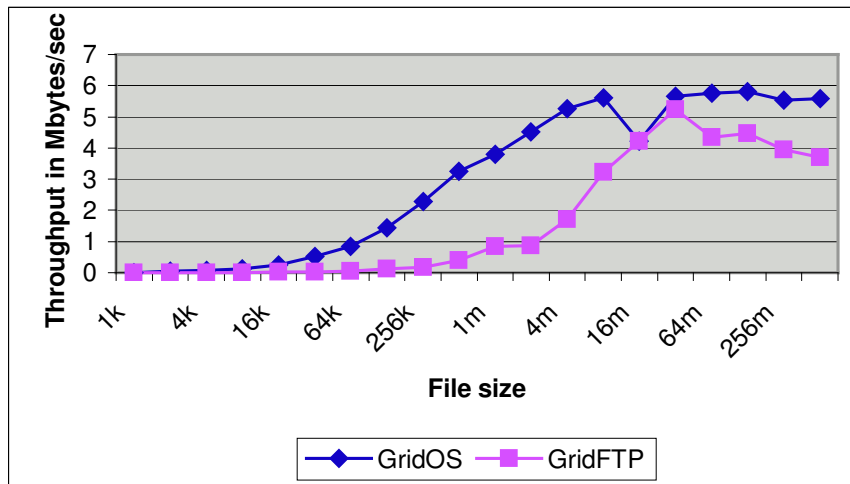


Figure 6–2: GridOS ftp vs GridFTP using globus-url-copy as client

## 6.2 GridOS vs Proftpd

In our first experiment, we compared the performance of the proftpd and GridOS ftp servers using the standard ftp client shipped with Mandrake Linux as the client. Results are shown in figure 6–1. This experiment show-cases the advantages of serving the files from within the kernel. GridOS consistently performs better than Proftpd for all file sizes.

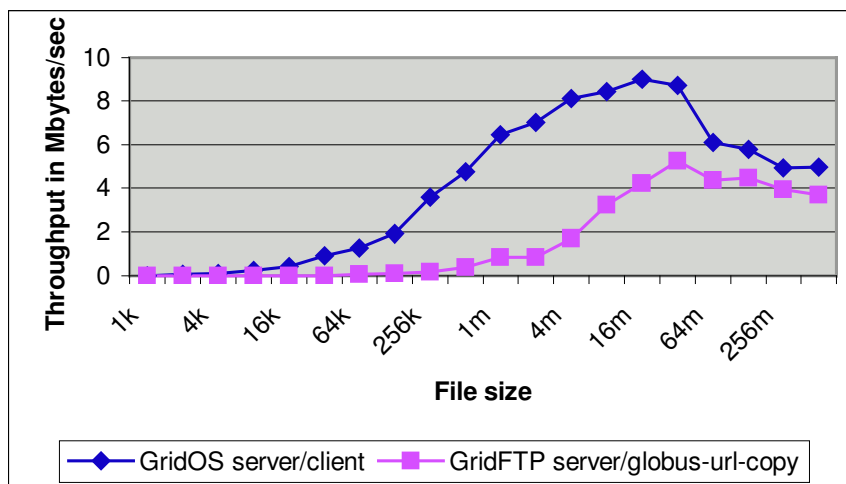


Figure 6–3: GridOS ftp server/client vs GridFTP server/globus-url-copy

### 6.3 GridFTP vs GridOS ftp server using globus-url-copy client

This experiment was done using the `globus-url-copy` client for a fair comparison. Results are shown in figure 6–2. GridFTP performs poorly for small files due to the overhead incurred in the performing security mechanisms. GridOS ftp employs the standard ftp password authentication for security.

We also conducted experiments using the standard ftp client instead of `globus-url-copy` for GridOS. Performance was poor compared to the performance obtained using `globus-url-copy`. `Globus-url-copy` is specifically designed for high-performance computing and has better buffer management. This led us to develop an in-kernel ftp client.

### 6.4 GridFTP server/client vs GridOS ftp server/client

In this experiment, `globus-url-copy` and `gridos-ftp-client` are used as clients for GridFTP and GridOS ftp server respectively. GridOS ftp client makes effective buffer management and is designed on the same lines as `globus-url-copy`. Results are shown in figure 6–3. Based on our experiments we observe that the performance drop for larger files is due to the overhead involved in disk writes.

## CHAPTER 7 GRID FILE SYSTEM

In the later days of my thesis work, I started working on development of a grid file system. The initial idea was to develop file system primitives in the operating system to provide facilities for higher level middleware. But, as it turned out, the file system primitives rightly belong to the middleware. In this chapter, motivation for the work, design and preliminary implementation results are provided.

In scientific disciplines using a grid, there is a growing need to access the data in the traditional file system semantics. To provide sharing of data generated from heterogeneous file systems in the grid environment, a logical hierarchical namespace with associated metadata that allows POSIX operations is required.

Currently, data middleware that are usually part of the general purpose grid middleware like Globus[2] and Legion[3] provide services for data management. In their earlier versions, middleware provided incompatible methods to access, manipulate and store the data. The Open Grid Services Architecture(OGSA)[7] defines the standard communication protocols and formats for building truly interoperable grid systems. The Open Grid Services Infrastructure(OGSI)[8] provides the specification for building various grid services envisioned in OGSA. These specifications have standardized grid services describing how they operate and interact. OGSI provides detailed WSDL specifications for standard interfaces that define a grid service.

The third version of Globus toolkit (GT3) provides implementation of OGSI specification and provides various OGSI-compliant basic services including managed job service, index service and reliable file transfer service (RFT). GT3 also provides

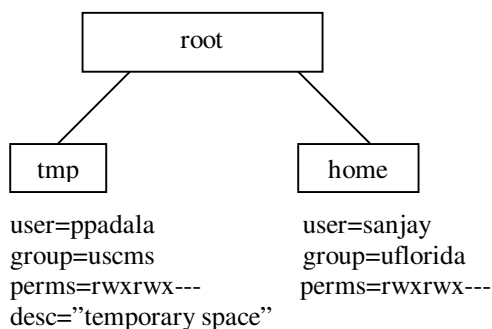


Figure 7-1: ufl.edu logical structure

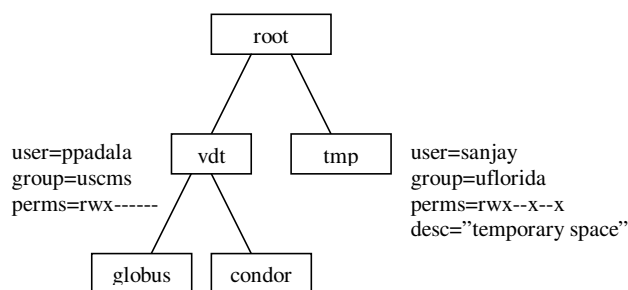


Figure 7-2: ucsd.edu logical structure

higher level data services including RLS (Replica Location Service). The data management services RFT and RLS provide the basic mechanisms for data management.

A grid application with complex requirements for data management often requires much more than what is provided by the basic services. Some of the features required include file or object level locking, automatic replica management and logical hierarchical name space.

We developed a file system service(FSS) for grids that provides a logical heirarchical name space that allows POSIX operations.

### 7.1 Motivating Example

Scientific applications in domains like High Energy Physics, Bioinformatics, Medical Image Processing and Earth Observations often analyze and produce massive amounts of data (sometimes of the order of petabytes). The applications access and manipulate data stored in various sites on the grid. They also have

to distribute and publish the derived data. Let's see a scenario of how a typical scientific application interacts with the data grid.

1. A physicist participating in a high-energy physics experiment would like to execute a high energy physics application.
2. The application requires various input files. It has to find the location of files using a catalog, index or database system where information about location of the file is stored. The application usually uses a logical file name(LFN) to index into the catalog and find the physical location.
3. The files may have to be replicated at various places in the grid for the application to find a nearby location to quickly access the file. The application may also have to prefetch the file to a designated site for execution of a job that requires the file at that site.
4. The jobs execute using the input files and produce derived data.
5. The derived data needs to be sent to various sites on the grid for usage by other scientists and for archival purposes.
6. Finally, the output data location has to be published in the catalog.

Using current middleware the above scenario requires the application to perform complex interactions with grid services. Also the application cannot perform the usual POSIX operations on a logical name. The application doesn't have full details of the file system and operations like *find all the sites with minimum 100MB temporary space* cannot be performed directly.

Grid File System Service (FSS) provides services that allow an application to be written in familiar POSIX semantics. Under the hood, FSS performs the above operations and can optimize the file access.

Each site participating in the grid exports a local file system structure to the world. This structure is a logical hierarchy as defined by the site administrator. The logical hierarchy can be mapped to physical resources in various ways. This

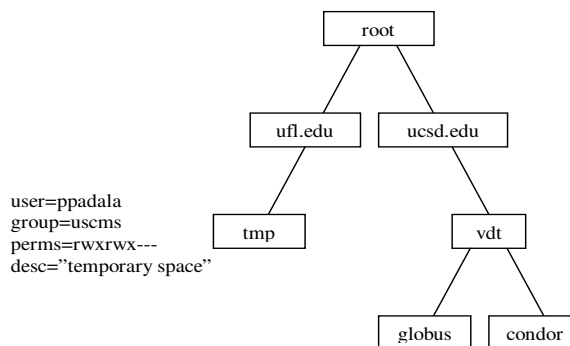


Figure 7-3: Find query result

logical to physical mapping is maintained in the local LRC (Local Replica Catalog). Figure 7-1 and 7-2 show logical mapping for two sites ufl.edu and ucsd.edu. Note that the logical directories do not have to follow the physical structure. For example, /tmp directory on ufl.edu may be mapped to /cise/tmp where as on ucsd.edu it might be mapped to /export/tmp. An application or user can create an application- or user-specific view of this global structure by using FSS services.

The following scenario clarifies the usage.

1. A physicist participating in a high-energy physics experiment would like to execute a high energy physics application
2. The application requests FSS to provide an application specific view on the global file system. For example, an application requests FSS to provide all the directories owned by user ppadala. This is similar to a `find . -type d -user ppadala -name "*"` command in traditional UNIX file systems.
3. FSS returns a logical hierarchy shown in figure 7-3.
4. Now, the application makes a query on the logical hierarchy to find the directory with description “temporary space” which returns /ufl.edu/tmp.
5. Application creates files in the temporary space using the logical name provided by FSS. Under the hood, the file is created in the specific site.
6. It closes the file and other applications can see the file.

7. Similarly, an existing file can be found by performing a query. For example, the user might be interested in all the temporary files created in various sites so that he can delete them.

Before we delve into the design of FSS, let us investigate the requirements of a grid file system. I am leading a survey for grid file system requirements for the GFS-WG (Grid File Systems Working Group) to understand various aspects of grid file systems. Here we delve into some aspects of the requirements of a grid file system.

## 7.2 Requirements

Investigating current middleware used in the data grids and the requirements for data management in various grid applications, we identified the following requirements.

- *Logical Hierarchical Name Space*: Since data is stored in different name spaces in the grid, it is essential to have a universal name space encompassing underlying naming mechanisms. In a data grid, data might be stored in a file, collection of files or in a database table. They may be replicated in various sites for efficiency. A logical hierarchical name space separates the application from the existing physical file system and provides a familiar tree file system structure. Replicas and Soft links complicate the structure.
- *Uniform Storage Interface*: The file system should provide a transparent, uniform interface to heterogeneous storage.
- *Data Access/Transfer*: Robust and flexible mechanisms for data access and transfer are required.
- *Replica Management*: Replication is the process of maintaining identical copies of data at various sites. The main purpose of replication is to maintain copies of the data at nearby sites thus increasing read performance. But, it introduces other issues like consistency management and life cycle management. The file system should provide basic replica management facilities like creation, update and deletion of replicas.
- *Latency Management*: Usually data for a grid application are distributed over various sites on the grid. A grid file system should provide mechanisms for managing the latency using mechanisms like streaming, disk caching, pre-fetching and bulk loading.
- *Metadata Management*: Metadata is the data about the data. There are various kinds of metadata described in detail later. A file system should

provide basic metadata management facilities and support for at least file level metadata. It should also support mechanisms through which additional metadata can be managed.

- *Optimization and Performance*: The file system should optimize the data access/transfer using various methods including data transfer prediction based on past information. Automatic replica management based on application access patterns is required.

## 7.3 Design

In the following sections, we detail the design of the FSS.

### 7.3.1 Logical Hierarchical Name Space

The logical hierarchical name space is the most important property of FSS. Each virtual organization or site running FSS can export a logical hierarchy to the world. The site administrator can decide on any arbitrary logical to physical mapping depending on his storage structure. The logical hierarchy is represented as a tree with nodes having attributes. The nodes have metadata associated with them. The metadata provides data about file size, creation time, access control information etc. More details are in section 7.3.4.

An application or user can send a query to local FSS and get a logical view as a result. This approach is similar to the query-based logical hierarchy proposed in the logic file system[30].

### 7.3.2 Data Access/Transfer and Replica Management

FSS builds on top of existing grid services *multirft*, *filestreaming* and *RLS*. Figure 7-4 shows this layered architecture. The multirft and filestreaming grid services are used for data access and transfer. FSS uses RLS for basic replica management providing more complex features like automatic replica management on top of it.

### 7.3.3 POSIX semantics

FSS provides a POSIX semantics for accessing the files. It provides the familiar open, read, write, close operations. An application must first request a

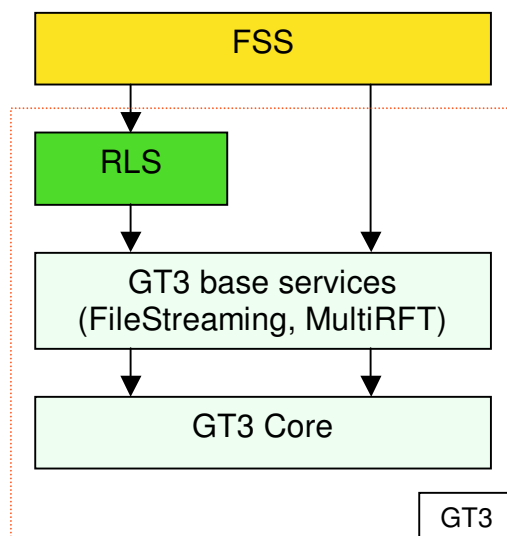


Figure 7-4: FSS architecture

logical view from the local FSS before it can do these operations. The logical view is cached by the local FSS with mappings to physical locations for fast access.

Details about the operations are given below.

- *open*: Open takes the file to open and the flags specifying whether the file should be opened as read-only, write-only, append or read-write. When a file is opened for writing, a write lock is placed on the file until the file is closed by the application. Open returns a unique handle that can be used to perform other operations. Note that the handle is unique only to the local FSS.  
The open request is propagated to the corresponding site FSS and the metadata corresponding to the particular file is updated. FSS creates a local disk cache for the file.
- *read*: The read operation can be done on an open handle. The filestreaming service is used to perform reads at a particular block. Reads are first performed on the local disk cache and if the data is not available in the cache, it is fetched from the remote site.
- *write*: Write operations write data to an open file. The writes are first done to the local cache and are propagated to the remote site via a background process. The filestreaming service is used to propagate writes. All replicas of the files are also updated.
- *close*: The close operation performs all pending writes, cleans the disk cache and updates the file metadata.

Directory operations like `mkdir`, `opendir`, `readdir`, `closedir` are provided for manipulating directories.

#### 7.3.4 Metadata

There are many kinds of metadata including file level metadata, storage metadata, access control metadata, application specific metadata etc. FSS attaches metadata with each node in the logical hierarchical name space. We defined the following metadata to be associated with each node in the logical tree.

- *File Level Meta Data*: This metadata describes the information about the file. This includes size of the file, creation, modification and access time, creator and locking information.
- *Access Control Meta Data*: This describes the permissions on the file. Currently, FSS supports traditional user, group, other permissions.
- *Application Specific Meta Data*: Scientific applications often associate various metadata including provenance and derivation information with the files. The `desc`(description) attribute shown in the previous examples is used to provide information about the directory.

There is a lot of potential for associating special purpose, application specific metadata that can improve the queries.

#### 7.3.5 Automatic Data Management

FSS can perform automatic data management depending on the file access patterns. For example, replicating a much accessed file (called hot-spot) at various sites improves performance and can be replicated automatically.

#### 7.3.6 Performance Optimization

FSS opens up lots of possibilities for efficient data management. The logical views for an application group can be maintained together and can be cached at the local FSS. It can also be statically stored at a central location like metadata catalog server.

Aggregation of files as a single file and Distribution of fragments of file provide mechanisms for efficient distribution of large data. Queries for finding the data can be optimized and executed by adding additional attributes to the logical hierarchy.

## 7.4 Building the Service

In this section, we describe our preliminary FSS design. Note that these descriptions are only preliminary and will be improved over time. Currently, we have defined description for basic open, read, write, close and getLogicalView operations. The GWSDL description is shown in appendix 9.

### 7.4.1 ServiceData

The servicedata will contain elements for specifying the number of managed files, number of open files and their status.

```
<gwsdl:serviceData name="managedFiles"
    type="xsd:int"
    minOccurs="1"
    maxOccurs="1"
    mutability="mutable"
    nillable="false">
    <documentation>
        number of managed files
        by the service
    </documentation>
</gwsdl:serviceData>
<gwsdl:serviceData name="fileStatus"
    type="fileStatusType"
    minOccurs="1"
    maxOccurs="1"
    mutability="mutable"
    nillable="false">
```

```
<documentation>  
    Status of the file  
</documentation>  
</gwsdl:serviceData>
```

#### 7.4.2 Notification

Many interesting scenarios can be implemented using the notifications provided by FSS. Notification for change of files, change of file metadata can be provided.

#### 7.5 Open Questions

There are many open questions to be answered for building the file system service. Below, We list a questions we are exploring.

- How can we share the logical views of an application among various FSS? What protocol should be used for this federation?
- What is the format for logical views that are returned to the application? We are thinking of returning the view in a XML format. How should one return a partial view if the results are too big?
- How can we efficiently manage the metadata? We are thinking of using the meta data catalog server that is available at each site for storing metadata of the files related to that site.
- How should we propagate the updates/writes to files and metadata. This is a big research question involving latency management and consistency protocols.
- What are the approaches for automatic replica management? We are investigating user-initiated, application-initiated, scheduler-initiated and automatic replica management.

## CHAPTER 8 FUTURE WORK

The work presented in this thesis is a first step towards a grid operating system that provides extensive, flexible services for grid architectures. Our next step is to implement all GridOS modules. We have plans to deploy GridOS in a wide area computing testbed. It can be used in nationwide test beds similar to GriPhyN (Grid Physics Network)[31] at the University of Florida. Such environments will enable use to evaluate GridOS modules more realistically.

There are many possibilities for high level middleware to exploit GridOS functionality. Globus core libraries can be ported to use GridOS functionality making them even more powerful. Interesting scenarios like interoperating between Condor and Globus job IDs using GridOS process management module are worth exploring.

The Grid File System opens up tremendous opportunities for efficient data management. Efficient data management techniques like optimal replica selection mechanisms can be developed using the Grid File System.

## CHAPTER 9 CONCLUSIONS

We have described a set of operating system services for grid architectures. These services make a commodity operating system like Linux highly suitable for high-performance computing. We have identified a common set of services that use grid software infrastructures like Globus, Legion, Condor etc. The services are developed as a set of modules for ease of use and installation. Our performance experiments show that high-performance can be achieved with GridOS modules. We have also described proof-of-concept middleware that uses GridOS facilities.

We presented the design for a grid file system that provides logical hierarchical name space, uniform storage interface, replica management and metadata management.

APPENDIX  
GRID FILE SYSTEM GWSDL DESCRIPTION

```
<xsd:schema targetNamespace="http://gridfs.base.ogsa.globus.org"
  attributeFormDefault="qualified"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="query">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="logicViewHandle">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="value" type="xsd:anyURI"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="openInput">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="handle" type="xsd:anyURI"/>
        <xsd:element name="flags" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

```
        </xsd:complexType>
</xsd:element>
<xsd:element name="fileHandle">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="value" type="xsd:anyURI"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="readInput">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="handle" type="xsd:anyURI"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="localTempFile">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="value" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="writeInput">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="inDiskCache" type="xsd:string"/>
            <xsd:element name="handle" type="xsd:anyURI"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

```

        </xsd:complexType>
</xsd:element>
<xsd:element name="writeOutput">
    <xsd:complexType name="value"/>
</xsd:element>
<xsd:element name="closeInput">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="handle" type="xsd:anyURI"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="closeOutput">
    <xsd:complexType name="value"/>
</xsd:element>
<xsd:simpleType name="status">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Open"/>
        <xsd:enumeration value="Reading"/>
        <xsd:enumeration value="Writing"/>
        <xsd:enumeration value="Locked"/>
        <xsd:enumeration value="Closed"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="fileStatusType">
    <xsd:sequence>
        <!-- refers to the logical name as known to this FSS -->
        <xsd:element name="fileName" type="xsd:string" />

```

```
        <xsd:element name="fileStatus" type="status" />
    </xsd:sequence>
</xsd:complexType>

</xsd:schema>
</types>
<message name="GetLogicalViewInputMessage">
    <part name="parameters" element="tns:query"/>
</message>
<message name="GetLogicalViewOutputMessage">
    <part name="parameters" element="tns:logicViewHandle"/>
</message>
<message name="OpenInputMessage">
    <part name="parameters" element="tns:openInput"/>
</message>
<message name="OpenOutputMessage">
    <part name="parameters" element="tns:fileHandle"/>
</message>
<message name="ReadInputMessage">
    <part name="parameters" element="tns:readInput"/>
</message>
<message name="ReadOutputMessage">
    <part name="parameters" element="tns:localTempFile"/>
</message>

<message name="WriteInputMessage">
    <part name="parameters" element="tns:writeInput"/>
</message>
<message name="WriteOutputMessage">
```

```
        <part name="parameters" element="tns:writeOutput"/>
</message>

<message name="CloseInputMessage">
    <part name="parameters" element="tns:closeInput"/>
</message>

<message name="CloseOutputMessage">
    <part name="parameters" element="tns:closeOutput"/>
</message>

<gwsdl:portType name="GridFileSystemPortType" extends="ogsi:GridService">
    <operation name="getLogicalView">
        <input message="tns:GetLogicalViewInputMessage"/>
        <output message="tns:GetLogicalViewOutputMessage"/>
        <fault name="Fault" message="ogsi:FaultMessage"/>
    </operation>

    <operation name="open">
        <input message="tns:OpenInputMessage"/>
        <output message="tns:OpenOutputMessage"/>
        <fault name="Fault" message="ogsi:FaultMessage"/>
    </operation>

    <operation name="read">
        <input message="tns:ReadInputMessage"/>
        <output message="tns:ReadOutputMessage"/>
        <fault name="Fault" message="ogsi:FaultMessage"/>
    </operation>

    <operation name="write">
        <input message="tns:WriteInputMessage"/>
        <output message="tns:WriteOutputMessage"/>
    </operation>
</gwsdl:portType>
```

```
        <fault name="Fault" message="ogsi:FaultMessage"/>
    </operation>
    <operation name="close">
        <input message="tns:CloseInputMessage"/>
        <output message="tns:CloseOutputMessage"/>
        <fault name="Fault" message="ogsi:FaultMessage"/>
    </operation>
    <!--serviceData descriptions -->
    <!--shown else where in this document -->
</gwsdl:portType>
```

## REFERENCES

- [1] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, California, 1999.
- [2] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
- [3] Andrew S. Grimshaw, William A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.
- [4] V. Huber. UNICORE: A Grid computing environment for distributed and parallel computing. *Lecture Notes in Computer Science*, 2127:258–266, 2001.
- [5] Ian Foster. What is the grid? A three point checklist. *Grid Today*, 1(6), July 2002.
- [6] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.
- [7] I. Foster, C. Kesselman, J. Nick, and 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 2002.
- [8] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, and D. Snelling. Open grid services infrastructure (ogsi) version 1.0. *Global Grid Forum Draft Recommendation*, July 2003.
- [9] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998.
- [10] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer-Verlag LNCS 1459, 1998.

- [11] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10<sup>th</sup> Symposium on High Performance Distributed Computing*, pages 181–195, 2001.
- [12] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002.
- [13] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor : A hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, pages 104–111, Washington, D.C., USA, June 1988. IEEE Computer Society Press.
- [14] Rajesh Raman, Miron Livny, and Marvin Solomon. Resource management through multilateral matchmaking. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, pages 290–291, Pittsburgh, PA, August 2000.
- [15] Steve J. Chapin, Dimitrios Katramatos, John Karpovich, and Andrew S. Grimshaw. Resource management in legion. Technical Report CS-98-09, Department of Computer Science, University of Virginia, February 11 1998.
- [16] Amin Vahdat, Tom Anderson, Mike Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, pages 52–64, July 1998.
- [17] Sudharshan Vazhkudai, Jeelani Syed, and Tobin Maginnis. PODOS — the design and implementation of a performance oriented Linux cluster. *Future Generation Computer Systems*, 18(3):335–352, January 2002.
- [18] Andrew S. Tanenbaum and Sape Mullender. An overview of the Amoeba distributed operating system. *Operating Systems Review*, 15(3):51–64, July 1981.
- [19] John K. Ousterhout, A. R. Cherenson, Fred Douglass, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.
- [20] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS-99)*, pages 78–88, New York, May 5 1999. ACM Press.
- [21] W. R. Stevens. *TCP/IP Illustrated, Volume 1; The Protocols*. Addison Wesley, Boston, Massachusetts, 1995.

- [22] J. Postel. RFC 793: Transmission control protocol, September 1981.
- [23] L. Kleinrock. *Queueing Systems: Theory*, volume 1. John Wiley and Sons, New York, 1975.
- [24] J. Semke, M. Mathis, and J. Mahdavi. Automatic TCP buffer tuning. *SIGCOMM 98*, pages 315–323, 1998.
- [25] Brian L. Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake, Jr. A network-aware distributed storage cache for data-intensive environments. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 185–193, Redondo Beach, CA, August 1999. IEEE Computer Society Press.
- [26] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.
- [27] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.
- [28] P. Tobin Maginnis. Design considerations for the transformation of MINIX into a distributed operating system. In ACM, editor, *Proceedings, focus on software / 1988 ACM Sixteenth Annual Computer Science Conference, February 23–25, the Westin, Peachtree Plaza, Atlanta, Georgia*, pages 608–615, New York, NY 10036, USA, 1988. ACM Press.
- [29] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pages 199–212, Berkeley, CA, June 6–11 1999. USENIX Association.
- [30] Yoann Padioleau and Olivier Ridoux. A logic file system. In *Proceedings of USENIX 2003 Annual Technical Conference*, pages 99–112. USENIX, June 2003.
- [31] Paul Avery and Ian Foster. The griphyn project: Towards petascale virtual-data grids. Technical Report GriPhyN-2000-1, University of Florida, 2001.

## BIOGRAPHICAL SKETCH

Pradeep Padala was born on August 22<sup>nd</sup>, 1979, in Srikakulam, Andhra Pradesh, India. He received his undergraduate degree, Bachelor of Engineering, computer science and engineering, from Motilal Nehru Regional Engineering College, Allahabad, India, in May 2000.

Pradeep worked in Hughes Software Systems from August 2000 to August 2001. He joined the University of Florida in Fall 2001 to pursue his master's degree. His research interests include distributed systems and operating systems with an emphasis on grid computing. More details about him can be found on his web site at <http://www.cise.ufl.edu/~ppadala>