A study in resource federation for e-Science

Yutaka Kawai

DOCTOR OF PHILOSOPHY

Department of Accelerator Science
School of High Energy Accelerator Science
The Graduate University for Advanced Studies

2012

Abstract

This research seeks to seamlessly support the infrastructure of distributed computing and storage through the development and study of a software-abstraction layer that interfaces to multiple Grid middleware and to new Cloud environments. Through this abstraction it is possible to sustain uninterrupted access to resources that is robust to the dynamic nature of those resources (compute nodes may fail, storage resources may go offline while a computation is being performed). We studied the software-abstraction layer and provided our Universal Grid User Interface (UGI) architecture for multiple kinds of Grid and Cloud middleware to support end users and application engineers. UGI is implemented based on A Simple API for Grid Applications (SAGA) and provides supplemental and extended functions that are not included in SAGA.

We demonstrated that job submissions can be executed in the UGI-based user environment with different Grid resources. We provided and verified a simple way to execute the jobs based on High Energy and Nuclear Physics (HENP) libraries. For file manipulation, we demonstrated that an application can access the different file-system middleware in the Data Grids. The application enables to handle pieces as completed files, even if a large file is cut up and the separated parts are stored on different Data Grids. We managed the files distributed in heterogeneous Data Grids by using a catalog service. The example demonstrated that an application can obtain the location information about the pieces of files distributed among different kinds of Data Grids, and then access the distributed files.

For applied tools and applications, we demonstrated a method to reliably manage files with Resource Namespace Service (RNS), a UGI-based Web application for Particle Therapy Simulation (PTSim), and an approach inspired by Ant Colony Optimization (ACO). Our method for reliably managing large files works on different kinds of Data Grids using RNS. The volume of digital data and the size of an individual file are increasing due to the introduction of high-resolution images, high-definition audiovisual files, etc. The reliable storage of such large files is becoming problematic with whole file replication as a failure in the integrity of the file is difficult to localize. Our method involves managing large files in Data Grids by splitting them into smaller units in a traceable manner and then managing the smaller units. The RNS catalog service contains EPR (Endpoint Reference) and metadata that describe the original locations as well as the checksum values. The example we shows how our Grid application can retrieve the actual file locations

and the checksum values from the RNS service.

Our second tool is a UGI-based Web application for PTSim. PTSim is a simulation system for particle therapy. The application of particle physics to the medical environment is one of the application areas that have a direct benefit to mankind. PTSim makes use of the Geant4 toolkit to simulate the passage of particles through the human body. It includes a Web interface that can be used by several collaborating medical particle therapy centers. The Web interface allows a non-Grid environment to be easily ported to Grid to take advantage of the additional resources.

Our last tool is for an approach inspired by swarm intelligence, ACO. Swarm intelligence is one of approaches to provide a fault tolerant and efficient means of transferring data in a dynamic environment. Swarm intelligence is inspired primarily by observations of the collective behavior of social insects in addressing complex distributed problems. The basic idea is that each member of the swarm has simple rules that govern its behavior, but the interaction among the members of the swarm can be used to tackle problems that are difficult to solve with complicated numeric methods. We investigate the problem of data distribution among a client and servers in a dynamic environment. We regard each download from a server to the client as a single member in a swarm. The member's behavior is simply to reliably download a data file. Each member can communicate with other members to allow the swarm to settle on the best set of servers to download the data from based on the current status of the environment. ACO is one of Swarm intelligence methods. We created a simulator following the ACO based approach and showed that our approach works well, providing a fault tolerant and efficient means of transferring data in a dynamic environment.

We can utilize the computing and storage resources with our implementation and solution. The challenges of today's researchers who need to collaborate with geographically distributed colleagues with distributed computing and storage resources can be overcome.

Acknowledgment

First of all, I appreciate my supervisor Takashi Sasaki. He gave me the opportunity to get into this research. He has not only helped me shape my research but also guided me in various aspects of my life as a researcher. Without him, this dissertation would not have appeared.

I am grateful to Go Iwai. He was my colleague in RENKEI project at KEK and also a committee member of this dissertation. He has supported me to work on papers, talks and presentation slides for academic conferences. Also, he helped me to set up my research environments in the beginning of my KEK life.

I would like to thank Hideo Matsuda who is another committee member. He leads the RNS development which I was working on a software adaptor for. He game me lots of discussions related to RNS and I was therefore able to improve my implementation.

I would also like to thank other committee members: Toshaiki Kaneko, Kento Aida, and Hiroyuki Matsunaga. I appreciated receiving their comments on my work and this manuscript.

I also thank to those who worked with me on the iRODS researches. Adil Hasan kindly suggested corrections against my wrong and vague parts in research aspects and also in English expressions. Francesca Di Lodovico and Yoshimi Iida suppoted to setup iRODS between KEK and QMUL. My implementaion used the iRODS testbed which is set up by them with Jean-Yves Nief from CC-IN2P3, Lyon, France and Mike Wan, Wayne Schroeder, Arcot Rajasekar and Reagan Moore from the DICE group.

I wish to acknowledge the valuable supports provided by Yoshiyuki Watase and Wataru Takase for the implementation of UGI and Web interfaces. I would like to thank members of Geant4 collaboration and members of PTSim development, especially, Takashi Akagi and Tomohiro Yamashita for using the DICOM data. It is also a pleasure to acknowledge the SAGA developer team led by Shantenu Jha and Andre Merzky for their valuable suggestions and support.

I was encouraged much by some members in the RENKEI project: Kenichi Miura, Kazushige Saga, Kiyoshi Yamada, Eisaku Sakane, Yoshiyuki Kido, Yoshikazu Tanaka, Hitohide Usami, Osamu Tatebe, Shin'ichirou Takizawa, Nobukazu Yoshioka.

I also thank to those who gave me precious comments and advices at CRC in KEK: Sou Suzuki, Kouichi Murakami, Jiro Suzuki, Atsushi Manabe and Fukuko

Yuasa. I would also like to thank to CRC staffs, Mitsune Arai and Yumiko Kimura for supporting my office environments.

Special thanks are expressed to Shannon S Jacobs for correcting the English language of my conference papers, journals, and this manuscript.

Finally, I greatly thank my family. I appreciate my wife, Junko, for her patience. She supported me at home while I concentrate on this study. Our children, Haruka and Wataru, bring me peace and comfort. My parents have given me financial and mental support for many years. Any of my success will be also theirs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

For efficient research we need to use information and communication technology effectively. Grid computing and Cloud technologies are leading examples of distributed processing technologies using the Internet. Grid computing has already been used widely since it was developed with a focus on the accelerator science field. However, for the present Grid computing technology, the development and operation of each kind of middleware varies widely among different countries or areas. For this reason, the interoperability among different kinds of middleware has become a major problem.

This research focuses on an infrastructure for seamless distributed computing studied and developed as a software-abstraction layer interface that can be used with multi-Grid middleware and new Cloud environments. The calculations and simulations in the accelerator science require many computing and storage resources. Even if one site is damaged due to a disaster and the computing resources are diminished, it is possible to prevent discontinuation of research by using the resources of other sites. If a unified procedure that can be easily used is available to handle the applications and data in the different sites and systems, then utilizing the distributed processing and storage resources on a global scale is more possible.

System complexity of Grid or Cloud computing is increasing but application engineers and system administrators need to modify their systems, to add extra services and to support users. The new required software libraries and interfaces are always requested. The troubles and problems of Grid systems are also reported by users everyday (e.g. the Global Grid User Support (GGUS) [1]) Then, the system-complexity increase impacts on their workload. Therefore, a client-based system that is no impact or changes on the server-side is required. Our software-abstraction layer is designed to be independent from any middleware. We finally found algorithms inspired by swarm intelligence[2] to obtain distributed data in an optimal way without any change or overhead on the middleware-side. Our software-abstraction layer and the client-based algorithms are good combination

to utilize resources in different kinds of Grid middleware.

1.2 Definitions

1.2.1 e-Science Definition

The UK National e-Science Centre (NeSC) [3] defines "e-Science" as:

> e-Science is "the large scale science that will increasingly be carried out through distributed global collaborations enabled by the Internet. " [4, 5]

According to the intentions of NeSC, e-Science typically has to provide to "access to very large data collections, very large scale computing resources and high performance visualization back to the individual user scientists."[4] They also say "The Grid is an architecture proposed to bring all these issues together and make a reality of such a vision for e-Science."[4]

Grid computing is one of the foundations to make e-Science a reality. However, we are studying the realistic situations involving today's Grid computing. We found that the current state of Grid computing is inadequate for e-Science in the truly intended sense. We first define the Grid and then consider its limitations in the next sections.

1.2.2 Grid Definition

A history of the Grid definition appears in Professor Foster's document "What is the Grid? A Three Point Checklist" [6]. Ian Foster et al. wrote an initial definition of the Grid in 1998 as:

> "A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [7]

They later refined the definition in the article[8] as:

> "Grid computing is concerned with coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations." [8]

The key concept is the ability to negotiate resource-sharing arrangements among a set of participating parties (providers and consumers) and then to use the resulting resource pool for various purposes. They defined this as a Virtual Organization (VO):

> "The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization."

Finally, Ian Foster provided three criteria for a Grid as below:

1) A Grid must coordinate resources that are not subject to centralized control.

2) A Grid must use standard, open, general-purpose protocols and interfaces.

3) A Grid must deliver nontrivial qualities of service (e.g., relating to response time, throughput, availability, and security) for co-allocating multiple resource types to meet complex user demands.

The Grid definition will be discussed in more detail in Section 2.1.

1.2.3 Resource Federation

The efforts of e-Science middleware providers have certainly resulted in more mature Grid technology in recent years. In the last 10 years, different nations or regions have developed and deployed different Grid middleware infrastructures for e-Science. However, each Grid middleware is still utilized as just a fundamental infrastructure and is far behind of the primary idea analogized with electrical power Grid.

Therefore, federating those resources on different Grids scattered around the world becomes the very essence of e-Science now. Today's international scientific collaboration requires the resource federation which provides shared hardware and software resources on the different kinds of middleware.

1.3 Related Difficulties and Problems

The current kinds of Grid middleware are variedly developed among different communities, regions, and countries. As examples, Globus [9] is developed in the US, EGEE [10] gLite [11] is developed by the European Organization for Nuclear Research (CERN) [12], and NAREGI [13, 14] (NAtional REsearch Grid Initiative) is created as the Japanese national Grid middleware by the National Institute of Informatics (NII) [15]. The Globus technology has led in the Grid computing area since 1997, when the first version of the Globus Toolkit [16] appeared. Most

of the other kinds of Grid middleware, such as gLite and NAREGI, also use the Globus technology in their architectures. However, each kind of Grid middleware still has a different interface.

Foster's criteria require that a Grid interface be general-purpose, but such an interface does not yet exist. The problem is that each kind of Grid middleware in use has its own specialized interface. Users encounter difficulties in handling these different kinds of Grid middleware. Each Grid relies on advanced middleware to interface between its resources and applications[17]. Users have to be aware of the underlying middleware layer and therefore they also have to make their applications run within the middleware infrastructures that they are using.

In the High Energy Accelerator Research Organization (KEK) [18], there are a number of computing systems and storage resources in different kinds of Grid environments. It is not easy for the users at KEK to switch to different middleware. Learning how to use different kinds of Grid middleware is a burden on the users whenever there are problems or when new middleware infrastructure appears. Also, recent scientific challenges require worldwide collaboration among researchers and sharing of their resources, such as computing systems, large amounts of distributed data, software, and knowledge.

Therefore it is an essential to provide a uniform architecture for application developers and to offer a high-level abstraction layer as a bridge between the middleware and applications.

1.4 Research Approach

This dissertation describes the required functions to handle the various kinds of Grid middleware with a unified interface. Today, we have several kinds of Grid middleware: Globus from the US, EGEE gLite that is mainly developed by CERN, NAREGI, the Japanese national Grid middleware, and so on. There are also some local batch schedulers such as PBSPro, Torque, and LSF (Load Sharing Facility) [19]. In terms of storage resources, there are different kinds of Data Grid middleware, including GridFTP (Globus) and The integrated Rule-Oriented Data System (iRODS) [20, 21] that was primarily developed in the US, Grid data farm (Gfarm) [22] that was developed by Tsukuba University in Japan, and others.

We tried to handle these different kinds of Grid middleware with our implementation so that our experimental implementation can be of practical use in actual research environments, according to the requirements of the researchers by joining Open Grid Forum (OGF) [23] workgroups. We especially joined an OGF working group of A Simple API for Grid Applications (SAGA) [24], created some adaptors for SAGA, and evaluated our and their prototypes. We contributed to familiarize the Japanese Grid middleware, NAREGI, in the SAGA working group with creating a NAREGI adaptor for SAGA.

With our implementation, we tried to clear the unified-interface benefits with

actual use cases through working with projects related to Grid middleware and Data Grids, and published its contributions. We worked with RENKEI [25, 26] (REsources liNKage for E-scIence) project to discuss issues about Grid middleware in terms of NAREGI. We shared the issues and benefits about Globus-based middleware with the members from TeraGrid [27] and the successor XSEDE (Extreme Science and Engineering Discovery Environment) [28]. We also shared those about gLite middleware with the experts in CERN and KEK. In terms of Data Grids, we worked with the members of iRODS, Gfarm, and the Resource Namespace Service (RNS) [29, 30] to share the cutting-edge technologies of distributed file system and metadata.

1.5 Achievements

Here are the achievements of this work:

1. A method to provide interoperability with different kinds of Grid and Cloud middleware.
2. A new architecture to utilize distributed resources.
3. The development of a new software interface to utilize the computing and storage resources of different kinds of Grid and Cloud middleware.
4. Example solutions using the new interface.

1.6 Organization of this Dissertation

This dissertation is organized as follows. In Chapter 2 we describe the background of this study and discuss related work to clarify the current problems when using distributed resources in different kinds of middleware. Also, we review previous and related work on the utilization of distributed resources.

Chapter 3 describes our software-abstraction layer, the Universal Grid User Interface (UGI). UGI with its various functions allows us to share Grid and Cloud resources as well as local resources.

In Chapter 4 we describe our implementation. The methods for job submission with different Grid resources are discussed in Section 4.1 and in Section 4.2 we show how to access distributed storage resources in different kinds of Data Grids. In Section 4.3 we show how to manage files distributed in heterogeneous Data Grids with a catalog system. We discuss performance evaluations and the methods for the software-abstraction layer in Chapter 5.

Chapter 6 deals with how to solve complex cases using our implementation. Section 6.1 describes a method for reliably managing files distributed in different kinds of Data Grids with RNS. This approach results in more reliability for

large-file replication since different sub-file units can be stored on different storage systems, thus reducing the risks due to hardware failures. Section 6.2 shows an example of UGI application that is a Web based user interface for Particle Therapy Simulation (PTSim) [31, 32]. The prototype of the Web interface allows users to easily request most of their job operations.

In the last part of the dissertation, Chapter 7, we consider an applied study to solve complex distributed problems. The demonstration showed that our Swarm Intelligence approach can find the optimum performance parameters in a real environment. This research has the potential to be used in different kinds of Data Grids with UGI.

Chapter 2

Background and Related Work

We face challenges in using different Grid resources with various kinds of middleware today. It is easy for a user to access the resources within a given kind of middleware, but there are significant interoperability problems if a user wishes to combine (or federate) resources from different Grid and Cloud providers. We need to ensure compatibility with our applications that are designed for our own middleware environments. Researchers are now geographically distributed due to the globalization of research activities and their middleware environments are different.

The problems can be broken down into three main areas:

- Job Submission: executing jobs and obtaining results
- File Manipulation: sharing files and managing the catalogs in distributed storage
- User Interfaces: submitting jobs and manipulating files with a unified interface

In this chapter we first define Grid computing and then present problems and related work in each of these areas.

2.1 Computing Resource Definitions

The distributed resources in the Internet can be classified into two general groups. One is computing resources for submitted jobs and the other is storage resources for manipulating files. To define the Grid, we discuss the submission of jobs using computing resources in this section. When considering how to use the distributed computing resources, there are three approaches to job submission:

1. Remote Execution of Applications
2. Batch Queuing Systems

Each of these approaches requires a more complex system, with Grid computing being the most complex. Here is a summary of each approach for reference.

2.1.1 Remote Execution of Applications

There are many different ways to execute commands or run programs on a remote server. For UNIX machines, examples include rsh, rlogin, telnet, and ssh [33]. Users log into a remote server and execute their programs on that server (Figure 2.1). This is the simplest method, but the degree of parallelism is restricted by the server.

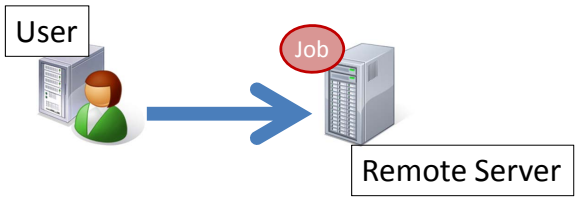


Figure 2.1: Job execution by remote server

2.1.2 Batch Queuing Systems (BQS)

A Batch Queuing System (BQS) can manage batch jobs and multiple worker nodes (Figure 2.2). The worker nodes are typically organized as a computing cluster. The BQS clients submit jobs to the BQS server and the BQS server schedules the submitted jobs to be executed on a selected worker node within the cluster. Users can increase the degree of parallelism compared to using a remote execution host.

2.1.3 The Grid

Figure 2.3 shows a rough design of the current implementation of Grid middleware. Each kind of Grid middleware basically consists of several main components. The following components are the examples of NAREGI:

- Super Scheduler (SS): Manages multiple CEs
- Computing Element (CE): Manages its own BQS
- Information System (IS): Manages resource information about the Grid
- Security System: Manages account and VO authentication for the Grid

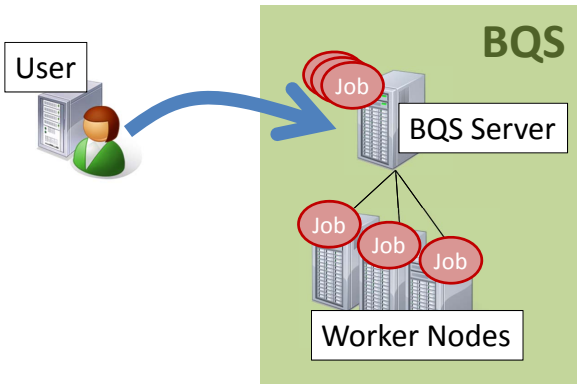


Figure 2.2: Job submission to Batch Queuing System

- Portal System: Provides environment tools for users of the Grid. (such as workflow tools, a Web interface, application-sharing tools, etc.)

The security and portal systems are closely related in many cases because authentications are required to use the environment tools. In order to discuss the file manipulation in a Data Grid, each Storage Element (SE) can be viewed as in Figure 2.3, linked to or replaced with a CE.

2.2 Job Submission

2.2.1 Job Submissions to Different Middleware

There are a number of computing resources with different kinds of middleware for job submissions in use today. Users need to ensure their backward compatibility because the applications of the users are specific to their own middleware environments. Table 2.1 summarizes the current situation regarding middleware that is being used, planned, or developed in several of the Virtual Organizations (VO) participating at KEK (as of May 2012). Some VOs use non-interoperable middleware in their resources.

VO	gLite	NAREGI	Gfarm	SRB	iRODS
ILC	Using	Planning	Planning		
Belle	Using	Planning	Using	Using	
Medical App.	Using	Developing	Planning		
Atlas	Using				
J-PARC	Planning	Planning	Planning		Testing

Table 2.1: Matrix between experiment and middleware.

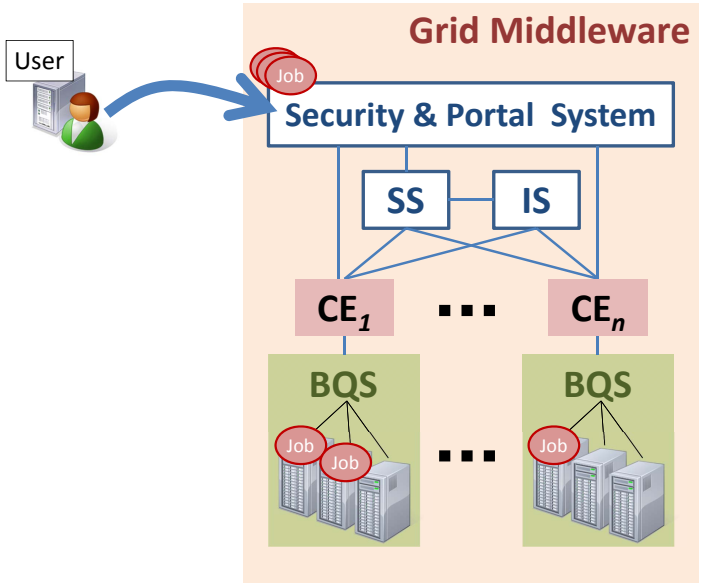


Figure 2.3: Job submission to Grid middleware

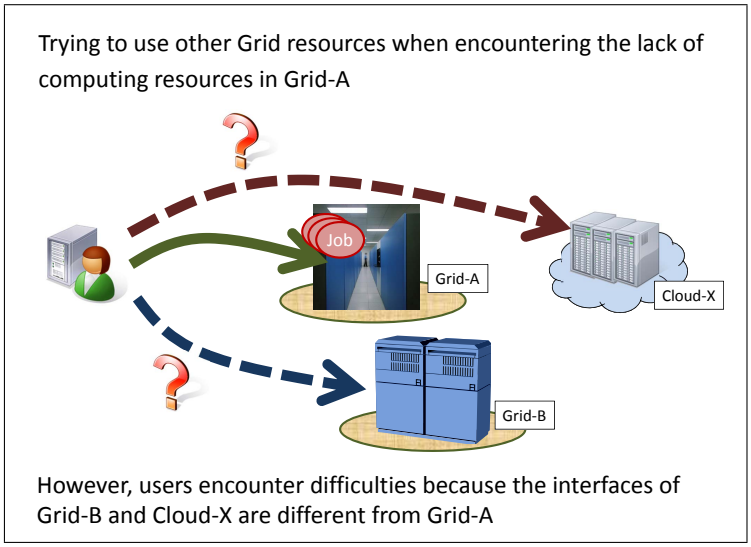


Figure 2.4: Difficulty of resource federation for job submissions

For the resource federation, here is a simple example involving different kinds of middleware: Grid-A, Grid-B, and Cloud-X (Figure 2.4). If the users usually use Grid-A but Grid-A is not operating correctly, they cannot continue working without the computing resources of Grid-A. In such a situation, they could use Grid-B or Cloud-X instead. However, there will be new problems in using such alternative resources because the interfaces of Grid-B and Cloud-X are different from those of Grid-A.

2.2.2 Related Work for Job Submission

There are several framework projects for Grid middleware to solve the above issues. Harald Gjermundrod developed the g-Eclipse [34] framework, which provides a workbench toolset based on the Eclipse architecture. This requires using the Eclipse GUI whenever a job is submitted. This is not suitable for researchers who mostly work with shell scripts or command interfaces. The gEclipse architecture is restricted to Eclipse-based plugin systems and its design depends on the middleware functions. There is a gLite middleware as an example, but some of the components required by gLite are scattered in various places in the architecture.

Erik Elmroth et al. created the Grid Job Management Framework (GJMF) [35]. The Job Control Service (JCS) is one component of GJMF, providing a functional abstraction of the underlying middleware and offering a platform- and middleware-independent job submission and control interface. GJMF supports the Globus and NorduGrid/ARC middleware. However, it is difficult to support other middleware because that would require changing the JCS components at the source level.

The Distributed Resource Management Application API (DRMAA) [36] provides a generalized API to facilitate integration of application programs. DRMAA is limited to job submission, job monitoring and control, and retrieval of the finished job status. These functions are close to the functions we need. In the SAGA specification [37] they note that "This API is also intended to incorporate the work of the DRMAA-WG [38]. Much of this specification was taken directly from DRMAA specification [36]". Therefore the experience of creating DRMAA is used to build the SAGA specification and DRMAA rendering in SAGA is possible. SAGA is a part of our software-abstraction layer (described in Section 2.4.2).

2.3 File Manipulation

2.3.1 File Manipulation among Different Data Grids

As shown Table 2.1, there are also a number of storage resources with different Data Grid middleware. Users again need to be careful about the backward incompatibilities. Again, we show a simple example that is similar to the situation in job submission. For the resource federation, we have different kinds of middleware: DataGrid-A, DataGrid-B, and Cloud-X (Figure 2.5). If users usually use DataGrid-A but DataGrid-A has some problems, they can use Grid-B or Cloud-X instead. However, they would still have difficulties due to the differences among DataGrid-A, DataGridB, and Cloud-X.

Other difficulties involve controlling the file locations in such differing Data Grid environments. Figure 2.6 shows two difficulties. One of them is that the path-name definition is different among the Data Grids. The other is that the file location must be shared among different users. We introduced RNS to manage files distributed in heterogeneous Data Grids. The detail of RNS is described in

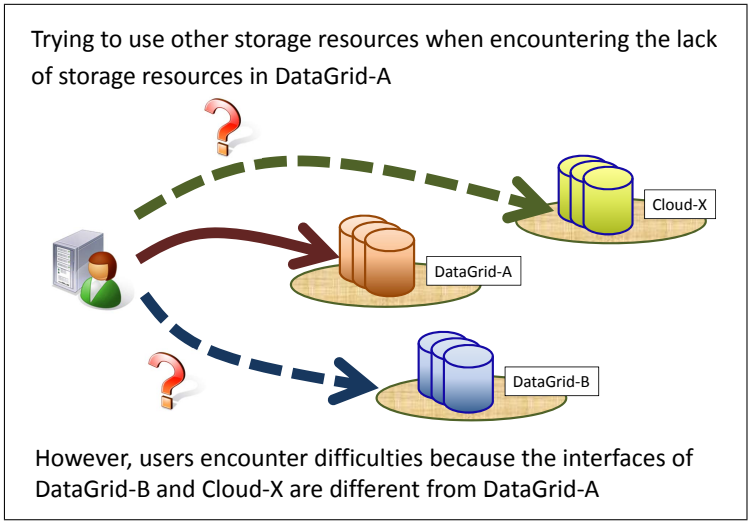


Figure 2.5: Difficulty of resource federation for file access

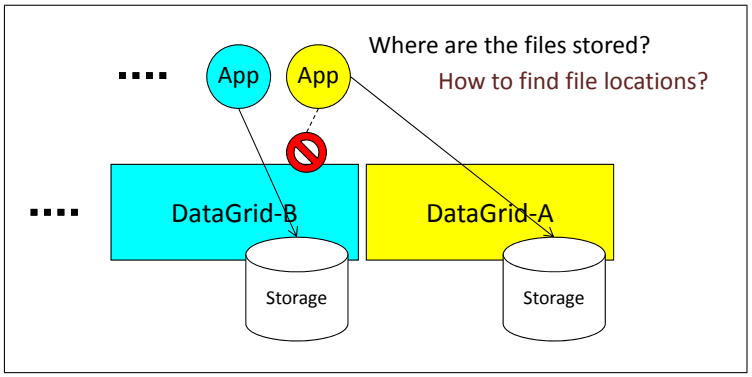


Figure 2.6: Difficulty of resource federation to control file locations

Section 2.3.2. The information about the physical file locations in our environment is managed as metadata entries in RNS.

2.3.2 Related Work for File Manipulation

RNS Overview

RNS, which was introduced in GFD101[29], "offers a simple standard way of mapping names to endpoints within a Grid or distributed network" [29]. As shown in Figure 2.7[39, 30], RNS provides hierarchical namespace management with name-to-resource mapping[39]. RNS has two fundamental types of entries, virtual directories and junctions. A virtual directory represents a non-leaf node in a hierarchical RNS namespace tree. A junction links a reference to an existing resource into the hierarchical RNS namespace. All compliant RNS implementations must have a valid WSAddressing[40] EPR.

The RNS application prototype is available from Osaka University[30] and

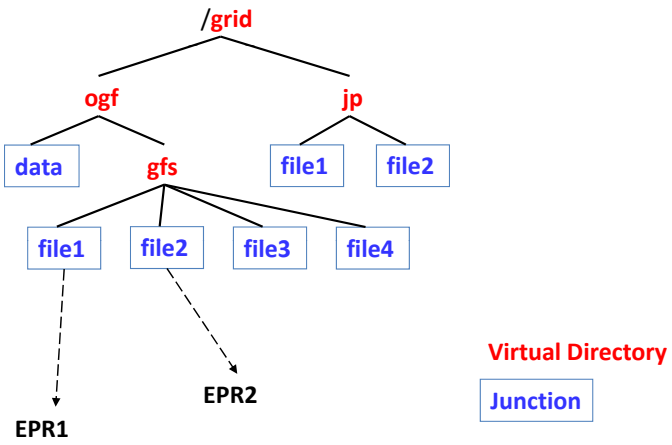


Figure 2.7: RNS: Hierarchical namespace management.

the University of Tsukuba[41] as a sub-project of the RENKEI project. The RNS servers and clients in the application communicate with XML messages using SOAP(Simple Object Access Protocol)[42] as defined in GFD101, and each RNS directory or junction entry can also contain its own XML messages as metadata.

As regards how to manage the file catalog in RNS, the RNS research group developed a metadata management system[43]. They assume the service sites become widely distributed and the number of the services is rapidly increasing. We can hierarchically search against all the RNS servers with a single query like a Domain Name System (DNS). RNS entries can spread over multiple servers. A logical file catalog in RNS can be built up among those RNS servers. RNS can manage a large-scale file catalog and static load distribution using multiple servers. In terms of dynamically updating the catalog among the servers, other researchers showed in their paper [44] but it causes increasing complexities of our test systems and environments. Therefore, in this study, we used RNS statically.

Our software-abstraction layer involves the RNS client implementation to manage files distributed in heterogeneous Data Grids. The API of the RNS prototype in our implementation is described in Chapter 4.

Other Catalog Services

There are several existing catalog services. The Storage Resource Broker (SRB) [45] has a uniform data access API and a metadata catalog. The SRB-derived iRODS inherits SRB's architecture and provides additional functions for rules and micro-services that can be customized by users[46]. OGSA-DAI(Open Grid Services Architecture - Data Access and Integration)[47] provides a client for metadata management with XML in the same way as RNS. Hermes[48] provides a desktop client for file transfer and data management with metadata. These services include mechanisms for resource discovery. However, the mechanisms are unique

to each service. To register the resources of a different service in the catalog of one service, additional adaptors must be used to access the other service, though OSGA-DAI tried to reduce these problems using an RNS approach[49, 50]. RNS can obtain an EPR using the SOAP protocol without those extra efforts.

AMGA (ARDA Metadata Grid Application)[51, 52] is a metadata catalogue service and part of the gLite middleware. AMGA provides a special SQL-like language to express queries but it has a limited number of operators[53]. In contrast, RNS accepts a general XQuery[54] to search the XML metadata.

Our work focuses on using RNS to minimize the special extensions needed by our Grid environments when managing distributed files.

Other Solutions for File Manipulation

Various projects are underway to reliably manage files and to aggregate different heterogeneous Data Grids or file systems for efficient data sharing. Here is a summary of work related to various aspects of our study.

Zeng Dadan et al. used different file systems for their Map-Reduce implementation [55]. They supported three file systems: a local file system, HDFS (Hadoop Distributed File System)[56] and the Kosmos File System[57]. However, they load and configure the interface within Hadoop, which limits their work to file systems supported by Hadoop.

Hamid-Reza Mizani et al. proposed VOFS (Virtual Organization File System) to hide the heterogeneity of aggregated file systems[58]. VOFS also has a metadata management service so that VOFS can handle traditional information (file size, last modification time, creation time, files in a directory, etc.) and VOFS-specific metadata. However, VOFS requires each registered file system to install a VOFS server.

Horst E Wedde et al.[59], Dan Feng et al.[60], and Yinjin Fu et al.[61] all discuss efficient metadata management. However, they are using only one or a few kinds of file systems.

Our solution with UGI is not limited to any specific environment. In addition, one only needs to modify a configuration file to switch file systems. UGI can handle the differences with the URL schema itself, thus our application can dynamically switch file systems. File systems do not need any customization in the solution and RNS can handle the resources of any kind of file system. Those are described in Section 4.2 and Section 4.3.

2.4 User Interface

2.4.1 Current User Interface to Control Differing Middleware

We need an interface to access middleware for job submission and file manipulation. Each current middleware has its own special interface and the interfaces are

```
Executable = ”test.sh”;
StdOutput = ”std.out”;
StdError = ”std.err”;
```

Figure 2.8: Job Description Example of gLite

```
<?xml version=”1.0” encoding=”UTF−8” standalone=”no”?>
<JobDefinition xmlns=”http://schemas.ggf.org/jsdl/2005/06/jsdl” xmlns:naregi=”http://www.
    naregi.org/ws/2005/08/jsdl−naregi−draft−02”>
  <JobDescription>
    <JobIdentification>
      <JobName>Program</JobName>
    </JobIdentification>
    <Application>
      <POSIXApplication xmlns=”http://schemas.ggf.org/jsdl/2005/06/jsdl−posix”>
        <Executable>test.sh</Executable>
        <WorkingDirectory>workdir</WorkingDirectory>
      </POSIXApplication>
    </Application>
  </JobDescription>
</JobDefinition>
```

Figure 2.9: Job Description Example of NAREGI

not compatible with each other. For example, Table 2.2 shows some examples of the differences between two kinds of middleware, gLite and NAREGI, in their job submission commands. There are not only differences in commands, but also job description differences between them. The differences of these job descriptions are shown in Figure 2.8 for gLite and Figure 2.9 for NAREGI. When we want to use both kinds of middleware, we must prepare job descriptions in both styles and manually switch them to execute the appropriate commands.

Function	**gLite**	**NAREGI**
Proxy Delegation	`glite-wms-job-delegate-proxy`	`naregi-signon`
Job Submission	`glite-wms-job-submit`	`naregi-job-submit`
Job Status	`glite-wms-job-status`	`naregi-job-status`
Job Cancellation	`glite-wms-job-cancel`	`naregi-job-cancel`
Job Retrieval	`glite-wms-job-output`	`naregi-std-print`

Table 2.2: Examples: Command differences between gLite and NAREGI.

For file manipulation, the situation is the same as for job submission. For example, Table 2.3 shows the differences between two kinds of middleware, iRODS and Gfarm, for storage commands. Beyond the command differences, there are also pathname differences between them. The differences in their pathname are shown in Table 2.4. We can specify any pathname in Gfarm, but iRODS requires

a $ZoneName$ in the beginning of its pathname.

Function	iRODS	Gfarm
List Files	`ils`	`gfls`
Copy File	`icp`	`gfrep`
Remove File	`imv`	`gfmv`
Concatenate File	`iget -`	`gfexport`
Create Directory	`imkdir`	`gfmkdir`

Table 2.3: Examples: Command differences between iRODS and Gfarm.

Middleware	Pathname Example
iRODS	`/tempZone/home/user1`
Gfarm	`/home/user1`

Table 2.4: Pathname Examples between iRODS and Gfarm.

To add new middleware for computing or storage, the operations and styles of the new middleware must be understood. Such extra work is basically wasteful for researchers, since the tasks of the job are quite often unchanged.

2.4.2 Related Work for User Interface

Using a Grid system involves using specialized commands and rules to access the middleware. The differences among these commands and rules can cause problems for application developers and users. Using a framework interface is one of solutions.

The Distributed and Unified Numerics Environment (DUNE) [62, 63] prepares abstract interfaces and a modular toolbox to solve particular equations with different kinds of Grid middleware. The DUNE framework consists of a number of modules: core modules and extra modules. Several Grid implementations can be used through the DUNE interface with the core modules. The extra modules allows to use other further Grid implementations. However, the DUNE interface does not apply published standards and it is especially designed for solving partial differential equations (PDEs).

The Open Cloud Computing Interface Core (OCCI) [64, 65] gives an abstraction to identify, classify, associate and extend distributed resources. However, OCCI is mainly used for Cloud systems because "OCCI was originally initiated to create a remote management API for IaaS model based Services" and the current OCCI is "suitable to serve many other models in addition to IaaS, including e.g. PaaS and SaaS."

SAGA is another architecture that provides a unified interface that conceals the differences among the different middleware infrastructures. A SAGA implementation is part of our software-abstraction layer, but SAGA does not have enough

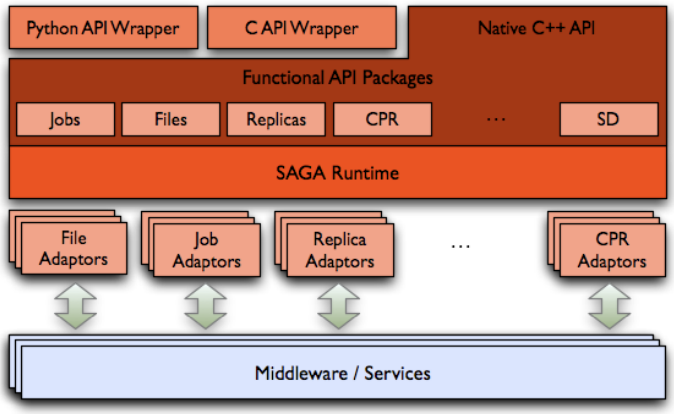


Figure 2.10: The design of SAGA implemented in C++.

functionality in such areas as authentication and job monitoring. We briefly describes SAGA in the next section. Supplemental and extended functions beyond SAGA are supported by UGI (Section 3.3).

SAGA Overview

SAGA [66, 24], is Open Grid Forum (OGF [23]) standard compliant software and is one of the realistic approaches to realize such an environment that is independent of the evolution of the middleware. SAGA is designed to be a bridge among the various kinds of Grid middleware (Figure 2.10 [67]). Once a SAGA adaptor for each kind of middleware has been prepared, the application developers only need the functional API without worrying about the specific features of the middleware. We can use any type of Grid resources if the appropriate SAGA adaptors are available. The SAGA community has released several SAGA core implementations[24]. The current SAGA C++ implementation supports a wide range of Grid middleware [68]. Python, C++ and Java APIs are currently available as the functional SAGA APIs. The APIs are used for the job modules, file modules, and replica modules.

SAGA APIs for Job Modules

Application developers can use these modules when invoking the APIs to submit jobs to the specified middleware. Application users need only specify the scheme to switch to different Grid middleware with the same job description. Table 2.5 shows some examples that application developers can invoke SAGA APIs for job submissions.

A SAGA job description has several attributes. The application developer can configure them one time and reuse the job description to submit jobs on other middleware infrastructures. The sample configuration of a SAGA job description

```
import saga
import sys

argvs = sys.argv
argc = len(argvs)

try:
    # Create a Job Description
    js_url = saga.url(argvs[1])
    job_caht = js_url.get_host()
    job_service = saga.job.service(js_url)
    job_desc = saga.job.description()
    job_desc.executable = './test.sh'
    job_desc.working_directory = '$HOME/work_dir'
    job_desc.candidate_hosts = job_caht
    # Submit a job
    my_job = job_service.create_job(job_desc)
    my_job.run()
except saga.exception, e:
    print "SAGA Error: ", e
```

Figure 2.11: Job execution example in Python interface.

is shown in Figure 2.11. Application users need only specify the job service (e.g. NAREGI or gLite) and modify a part of job description to switch it to another service.

SAGA APIs for File Modules

Application developers can use the file-module APIs to use file systems via SAGA. The application users need only specify the scheme and logical path to switch to some other file system middleware. Table 2.6 shows some examples that the application developers can call SAGA API to use the Data Grids.

SAGA APIs for Replica Modules

We also need to control metadata via SAGA. The logical file and the logical directory in the SAGA replica package allow us to handle the required metadata. Application developers can use these APIs to handle metadata for logical files and

SAGA API	Function
`saga::url::url()`	Specify job service (e.g. NAREGI or Torque).
`saga::job::description::description()`	Create a job description.
`saga::job::service::create_job(description)`	Create a job with description.
`saga::job::job::run()`	Submit a job.

Table 2.5: Frequently invoked APIs in SAGA job module.

SAGA API	Function
`saga::url::url()`	Specify directory or file location.
`saga::filesystem::directory::open(file)`	Open a *file* in the directory.
`saga::filesystem::file::read()`	Read the file.
`saga::filesystem::file::write(*buffer)`	Write *buffer* data to the file.
`saga::filesystem::file::get_size()`	Get file size of the file.
`saga::filesystem::file::copy()`	Copy the file.

Table 2.6: Frequently invoked APIs in SAGA file module.

directories. Table 2.7 shows some examples of application developers calling the SAGA APIs to handle metadata for the logical file and directory.

SAGA API for replica	Function
`saga::replica::logical_file::add_location(url)`	Add an *url* as a replica location to the logical file.
`saga::replica::logical_file::list_location()`	List the locations in the location set.
`saga::replica::logical_directory::set_attribute(key, value)`	Set a *key*(attribute) to a *value*.
`saga::replica::logical_directory::set_vector_attribute(key, values)`	Set a *key*(attribute) to an array of *values*.
`saga::replica::logical_directory::get_attribute(key)`	Get an attribute value.

Table 2.7: Frequently invoked APIs in SAGA replica module.

Chapter 3

Design of Abstraction Layer

This chapter describes a software-abstraction layer interface, the Universal Grid User Interface (UGI), to control the resources of different kinds of middleware. Today's international scientific collaboration requires the resource federation which provides shared hardware and software resources from various kinds of Grid and Cloud middleware. One of the solutions involves a unifying interface between the users and the middleware. We designed and implemented a UGI that provides a seamless environment for end users of such remote resources (Grid or Cloud resources) with their local resources. The UGI functions include job handling, manipulating files, general file cataloging, and monitoring jobs. UGI includes the SAGA (Simple API for Grid Applications) architecture and external components that are not supported by SAGA. Our prototype UGI implementation provides a Python API, a command line interface, and a Web interface.

3.1 Common Interface Solution

Current kinds of Grid middleware conform to most of Foster's criteria (Section 1.2.2) but there are still some issues. A general-purpose interface is one of his criteria, but has not been fully realized. Each of the current Grid middleware systems has a special interface based on its own architecture and none of the interfaces is designed as a generalized interface. The next level of interfaces should resolve the challenges of the interface differences among the Grid middleware and make simultaneously available all of the different resources in Grid middleware (Figure 3.1). The target of this dissertation is to solve these interface problems.

3.1.1 Job Submission with Common Interface

Figure 3.2 shows that a common interface enables the users to easily access different resources. Local clusters can also be used via the common interface. In addition, users can use all of the resources simultaneously. Such a situation can be described as a resource federation for job submissions.

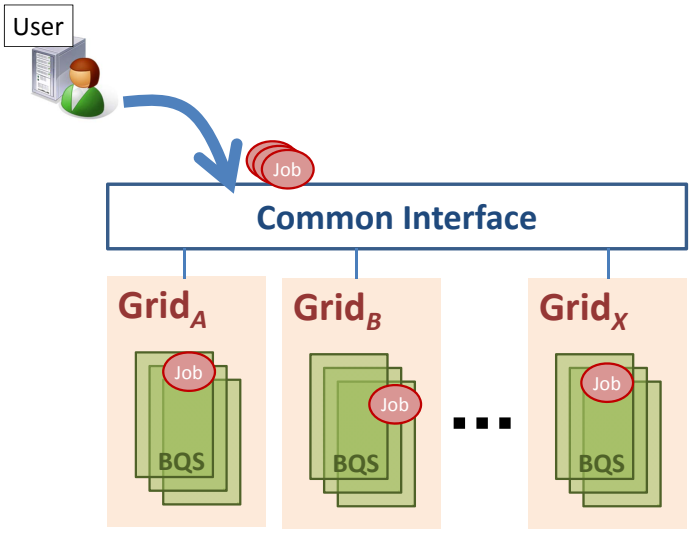


Figure 3.1: Submitting jobs to different Grids via Common Interface

3.1.2 File Manipulation with Common Interface

Preparing a common interface is also beneficial for file access. Figure 3.3 shows that the common interface allows the users to easily access different kinds of storage. Local file systems can be also used via the common interface. Obviously, users want to use all of the kinds of storage simultaneously.

We need to address two difficulties when controlling the file locations in different Data Grid environments, as mentioned in Section 2.3.1. RNS can provide a unified namespace among the Data Grids and its information can be shared among different users. RNS was introduced into the common interface to manage files distributed in heterogeneous Data Grids (Figure 3.4). The information about the physical file locations in our environment can be managed by the common interface. This makes it easier to share the metadata about each file between different Data Grids. Then users can access all of the available storage simultaneously and benefit from the catalog services for different kind storage resources.

3.1.3 User Interface for Interoperability

The common interface is a key component to solve the difficulties of job submission and file manipulation among different kinds of Grid and Cloud resources. One of the implementations concerns is where to place a software-abstraction layer for the common interface. The design objective of the software-abstraction layer is to hide the complex treatment of middleware from users and to provide them with seamless access to local, Grid, and Cloud resources.

Figure 3.5 shows the architecture of the software-abstraction layer with a Python layer and an Adaptation layer. Users prefer to use easy interfaces such

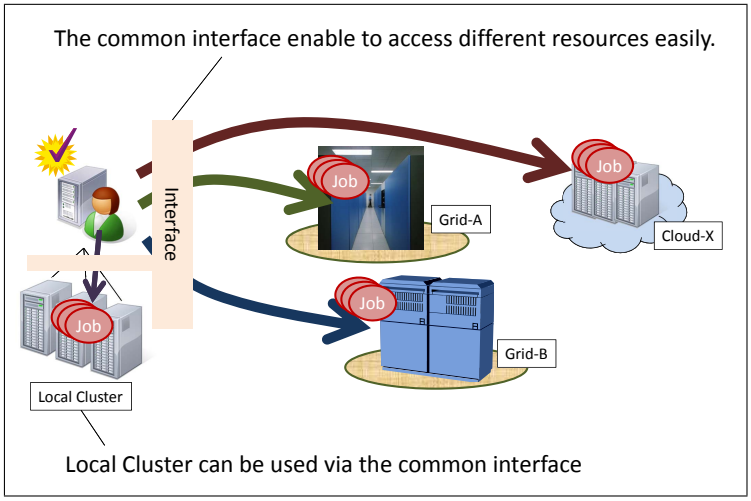


Figure 3.2: Place a common interface for job submissions

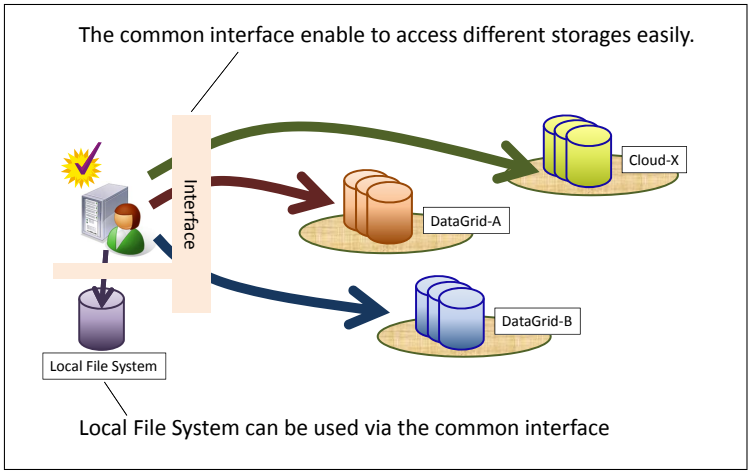


Figure 3.3: Place a common interface for file access

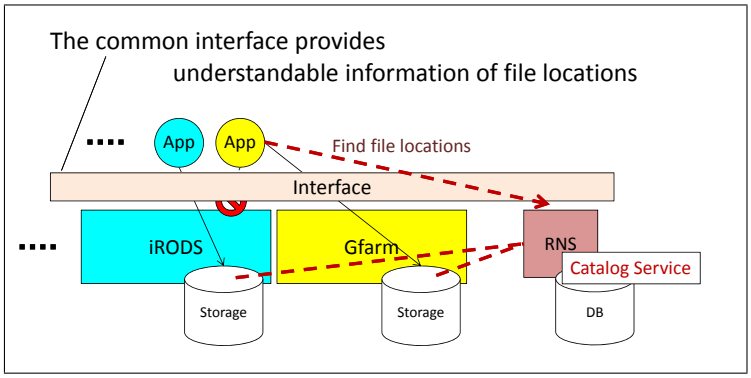


Figure 3.4: Place a common interface with RNS for file locations

as shell scripts, shell commands, and Web interfaces. Therefore the Python layer needs to provide at least a Python API, shell commands, and a Web Interface. The

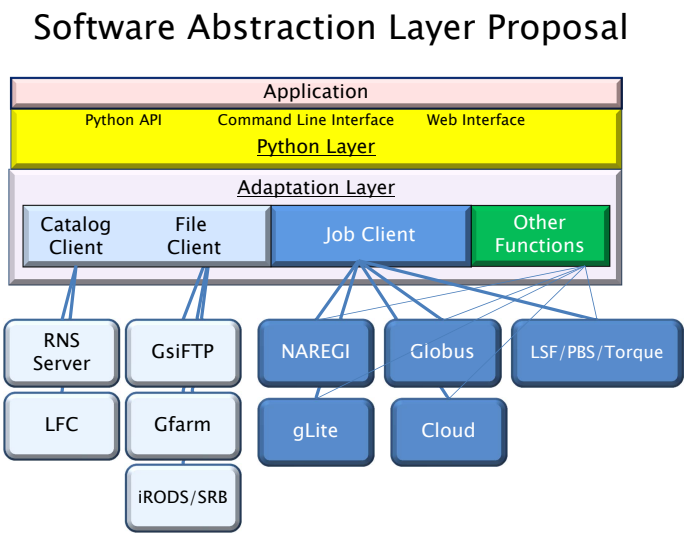


Figure 3.5: Implementation proposal for a software-abstraction layer

reason for adopting Python for the API is its popularity with scientists for quick development of custom applications for their fields of study. The Adaptation layer allows all of the clients to handle any middleware and consists of job clients, file clients, and catalog clients. The Adaptation layer is also flexible to include other functions for extra tasks such as authentication and monitoring.

In our currently supported environments, jobs are submitted to five kinds of middleware: NAREGI, Globus, gLite, PBSPro, and Torque. To manage files, we use three kinds of Data Grids: iRODS, Gfarm, GridFTP. The information about the physical file locations in our environment is managed as metadata entries in RNS. The abstraction layer helps application developers in building real applications for the end-user scientists.

The implementations of the software-abstraction layer must be easily maintainable and expandable for new kinds of middleware.

3.2 UGI Design

We designed a high-level user interface, UGI, and Figure 3.6 shows the architecture of the software building blocks. UGI is implemented based on SAGA and also provides supplemental and extended functions beyond SAGA. The UGI-based Web interface with the various functions can share Grid and Cloud resources as well as local ones. In order to share scientific resources among collaborators, we have to cope with different user interfaces to these different kinds of Grid middleware. We adopted SAGA to span the different Grid environments. SAGA aims to address this heterogeneity and currently provides working implementations in C++ and Java.

The current design of UGI functions includes job handling, file manipulation,

general file catalog, and monitoring. It runs on a host with SAGA C++ core libraries, adaptors, and client software necessary to use the Grid middleware and Data Grids.

For job handling, UGI supports multiple job submissions to various Grid kinds of middleware infrastructure and local batch systems at the same time using same job submission scripts. The job load sharing can be controlled according to the availability of resources. The end users' applications handle the results and data files according to their own work flow designs, which may call for such as procedures as chaining jobs, post processing, and graphic display.

UGI provides file manipulation functions such as copy, remove, transfer, and catalog registration. These functions accept various file storage protocols within the same API: Local file system, GridFTP, Gfarm, and iRODS. The file catalog is a crucial facility for sharing large amounts of distributed scientific data. For sharing files among different Grid middleware, a middleware-independent file catalog system is needed. UGI adopted the OGF standard RNS as its file catalog. We collaborated with Osaka University and the University of Tsukuba to implement the RNS as a general-purpose file catalog system. It provides a tree structure of the name space with virtual directories and junctions. Each junction has an EPR. Both virtual directories and junctions can contain metadata about the files or directories. The metadata can be queried to find appropriate files to use from a large archive of scientific data.

For monitoring a large number of jobs dispatched in different kinds of Grid environments, we introduced a lightweight database. The database can store not only job status data, but also job-related information such as job parameters, analysis conditions, output file locations, etc. Usually the status transition of jobs submitted to Grid middleware is delayed due to the propagation time from the middleware to the client. In UGI, a dispatched job can update the database by itself through a XML-PRC [69] mechanism.

3.3 SAGA Implementation

SAGA Adaptors

SAGA is a part of our UGI implementation. We implemented the required software modules to internationalize NAREGI. Actually, we developed the SAGA adaptors shown Table 3.1, in compliance with the SAGA specification, as standardized within the OGF[66, 70].

Other SAGA adaptors for other kinds of Grid middleware have been developed in other countries. Table 3.2 shows the currently available SAGA adaptors [68].

Through working on the SAGA adaptor implementations, we found that SAGA has some limitations in addressing our requirements. We cannot monitor the status of each job in real time via SAGA, SAGA does not support different kinds of authentication processes, and SAGA cannot handle directly copying files among

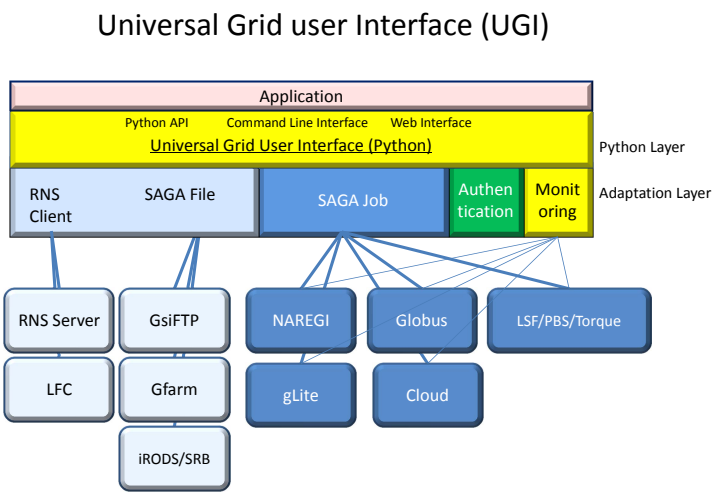


Figure 3.6: Architecture of Universal Grid Interface

Name	Full Name	Middleware
SNA	SAGA NAREGI Adaptor	NAREGI
STA	SAGA Torque Adaptor	Torque
SPA	SAGA PBSPro Adaptor	PBSPro
SIA	SAGA iRODS Adaptor	iRODS
SGFA	SAGA Gfarm Adaptor	Gfarm
SRA	SAGA RNS Adaptor	RNS

Table 3.1: SAGA adaptors developed in KEK.

different Data Grids. We describe these SAGA limitations in the next.

SAGA Limitations addressed by UGI

There are four problems in the current SAGA framework. First, there is a need to monitor jobs in real time. We cannot get real-time job status using the SAGA approach because the current system is designed to poll each job for its status. The number and complexity of jobs in recent research projects is increasing so we need to obtain the status of each job in real time. A UGI client can attach an epilogue script to report the job status via XML-RPC so that the job status can be monitored in real time.

Another problem is that the authentication for each kind of Grid middleware is different. SAGA supports only X.509 certificate [71] authentication by following the Globus procedure. NAREGI is one kind of Grid middleware that requires special commands that are not same as the Globus GSI (Grid Security Infrastructure) [72] authentication commands. We cannot use SAGA for the NAREGI authentication, and therefore we need a unified interface for any kind of Grid authenti-

SAGA Adaptor	Description
SSH	job submission and file transfer via SSH and FuseFS.
Globus	access to Globus Gram, GridFTP and Globus RLS resources
Condor	job submission to resources managed by Condor/Condor-G
LSF	job submission to a Platform LSF scheduler
X.509	X.509 certificate handling
BES	job submission to resource managers that support OGSA BES
EC2	job submission to Amazon EC2 compatible Cloud services
GridSAM	job submission to OMII GridSAM resource managers
gLite	job submission to gLite CREAM computing elements
HDFS	interaction with the Hadoop distributed filesystem(HDFS)
DRMAA	job submission to resource managers that support DRMAA

Table 3.2: SAGA adaptors developed by other contributors.

cation.

The next problem is the difficulty of file transfers among different kinds of Data Grids. SAGA can allow some file manipulations that are restricted to file systems using the same system. For example, when we want to copy a file from an iRODS path X to an iRODS path Y, SAGA can start this copy operation with the function:
`saga.filesystem.copy(`X,Y`)`
However, we cannot directly copy a file from iRODS path X to Gfarm path Z, because iRODS and Gfarm use different scheme in their SAGA URLs. Therefore, we need to prepare some third storage area for temporary space while transferring files between different kinds of middleware. UGI can directly copy these files with the function:
`ugi.file.transfer_copy(`X,Z`).`

The fourth problem is the developmental complexity of SAGA adaptors. Since each SAGA adaptor needs an API for each kind of Grid middleware, it requires several man-months for each implementation and the developer must understand the unfamiliar Grid middleware and its API specifications. UGI is written in Python and a new UGI adaptor can be created easily, because each adaptor just calls the commands for each kind of Grid middleware. This approach to easier implementation trades off some speed of development for performance.

3.4 UGI Functionalities

3.4.1 Job Handling

For a single job submission to a particular kind of Grid middleware, it is straightforward to write a script using Python. We introduced a Python object for multiple job submissions (with parameters) to a Grid middleware resource at different sites. The attributes of the object include the name of the middleware to be used, the site name, the number of jobs sharing the load, and the path of the job script.

The Python API for task submission is

```
ugi.job.submit(list_of_task)
```

which calls for simultaneous job and task submission to different kinds of Grid middleware. A typical shell command could be

```
ugi-job-muitiplejob-submit
```

with the arguments (middleware, site, njobs, path) providing the needed task attributes.

3.4.2 File Manipulation

Each Grid middleware has its own file storage system with its specific protocol for access. For multiple Grid environments, we need to hide these differences as much as possible. The available file protocols are local file, GridFTP, Gfarm, and iRODS. A typical example using this API is a file listing in a directory URL such as

```
ugi.file.ls(URL,options)
```

which returns a list object of file names. The URL accepts:
`file://...`,
`gsiftp://...`,
`gfarm://...`, and
`irods://...`
with the option: `-l` for a long listing. It also accepts an RNS catalog tree path such as `/rns/kek/ilc`. For files registered in RNS, there are additional options to get information about the registered files: `-u` (get the URL of the file), `-t` (get the transfer URL), and `-query` (a metadata query).

File transfers and copies are main functions in file manipulation. The SAGA scheme cannot support direct file copying between different storage protocols. The UGI API of `ugi.file.copy()` accepts different protocols for source and destination URLs by wrapping the proper commands to copy each file to or from its local file storage.

The RNS server running on a separate host stores the file catalog in a database (where the current implementation uses Apache Derby [73]). UGI obtains access to the catalog tree through the FUSE (Filesystem in Userspace) [74] mount mechanism to avoid the overhead of the client Java Virtual Machine (Java VM). The UGI API for catalog manipulations include functions such as
`ugi.file.register()`,
`ugi.file.unregister()`,
`ugi.file.replicate()`, and
`ugi.file.transfer-register()`.
UGI has command line interfaces corresponding to all of these API functions.

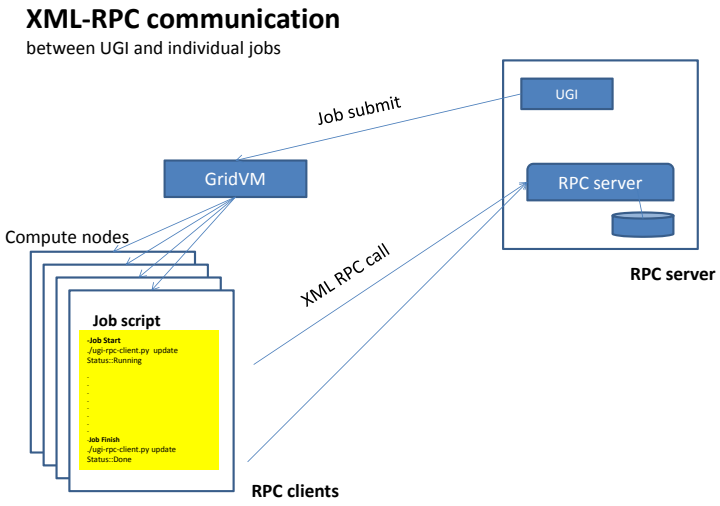


Figure 3.7: UGI monitoring mechanism

3.4.3 Monitoring

A SQLite [75] database is used as a lightweight database for storing the statuses of jobs dispatched to multiple kinds of Grid middleware as well as the job parameters and additional user-defined job information such as the output file location. UGI provides a useful database access API and commands. The database information is updated by each job using the API or commands in the job script. In order to access the database from running jobs, an XML-RPC server and client mechanism is used. The server runs on the local host and waits for client commands invoked by the job script. The job scripts includes the following:

```
./ugi-rpc-client.py ugi-mon-job-update Status::Running
```

or

```
./ugi-rpc-client.py ugi-mon-job-update Status::Done
```

Figure 3.7 shows diagram of XML-RPC communication between UGI side and jobs on the compute nodes. GridVM [76] in the figure is one of Grid-middleware components. In the case of NAREGI, GridVM should monitor and control resources and jobs per site. GridVM also should do the same per node in a site. RPC server and client can communicate each other through GridVM. RPC server is listening a specific port number (e.g. 24999) for client call. Then, the server receives job statuses from the client and updates the database. As an RPC client, a client script is staged-in in each job. The client sends user's command and its arguments. We can easily specify the statuses, "Status::Running" and "Status::Done", anywhere.

3.4.4 Authentication

User authentication may differ on each middleware. gLite and Globus uses the X.509 certificate authentication which requires a command, `grid-proxy-init`. We can also use the command, `voms-proxy-init` to generate a proxy with the VOMS (Virtual Organization Membership Service) [77] information. NAREGI Command Line Interface (NAREGI-CLI) requires a `naregi-signon` command to access NAREGI-Portal server. This mechanism involves MyProxy[78] technologies, thus NAREGI allows to use the commands of MyProxy to generate a proxy. In the case of LSF as one of BQS examples, it has a authentication command `kinit`. As an example of Data Grids, iRODS can serve both password and GSI authentications. iRODS requires a command `iinit` to use the password authentication.

As can be seen in the previous paragraph each middleware tends to use its own authentication commands. UGI uses a practical approach to provide a uniform interface to the different commands using a simple script, such as

```
ugi-cert-init.sh
```

which is an integrated command containing the middleware commands. In the current implementation the script supports:

- Globus Proxy Issue for Globus
- VOMS Proxy Issue for gLite and NAREGI
- VOMS Proxy Register to MyProxy Server
- `naregi-signon` for NAREGI
- `kinit` for LSF
- `iinit` for iRODS

This script can be easily expandable for other authentication commands and procedures.

Chapter 4

Implementation

The current available Grid middleware environments vary among communities, regions, and countries. Traditionally, using middleware requires using its own specific commands and rules. This chapter describes our implementation to handle the difference. Our computing and storage environments in this study are mainly based on Grid middleware and Data Grids, thus this chapter does not really discuss using Cloud services. Our implementation is easily applicable to Cloud services because all we need is to prepare adaptors for the services (e.g. Amazon EC2 adaptor shown in Table3.2).

4.1 Job Execution in Multi-Grid Environments

Under the multi-Grid environments, we are required to execute jobs and to define job descriptions following specific commands and rules of each Grid middleware. To manage the differences, we tried to deploy some applications which can work across the available middleware infrastructures. For example, in the High Energy and Nuclear Physics (HENP) community in Japan, the Computing Research Center (CRC) at KEK maintains the computing infrastructure for HENP (e.g. the central network services, software services, and so on). KEK is involved in RENKEI and studying to efficiently use both Grid and non-Grid resources, particularly for HENP users. Our applications work well for this community even in multi-Grid environments for end users working on such projects as Belle [79], ILC [80], and medical physics.

4.1.1 Setup Demonstration

Our software-abstraction layer, UGI, can mask the differences between different kinds of Grid middleware. This section describes the use of two types of middleware. One is NAREGI, and the other is Torque. Our demonstration shows how easy it is to deploy our application examples with different types of middleware. This chapter describes our software development using different middleware infrastructures, comparing the UGI environment with the traditional approach.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<JobDefinition xmlns="http://schemas.ggf.org/jsdl/2005/06/jsdl" xmlns:naregi="http://www.
    naregi.org/ws/2005/08/jsdl-naregi-draft-02">
  <JobDescription>
    <JobIdentification>
      <JobName>Program</JobName>
    </JobIdentification>
    <Application>
      <POSIXApplication xmlns="http://schemas.ggf.org/jsdl/2005/06/jsdl-posix">
        <Executable>test.sh</Executable>
        <WorkingDirectory>workdir</WorkingDirectory>
      </POSIXApplication>
    </Application>
    <Resources>
      <CandidateHosts>
        <HostName>nrg04.cc.kek.jp</HostName>
      </CandidateHosts>
    </Resources>
  </JobDescription>
</JobDefinition>
```

Figure 4.1: An example of WFML script

```
#! /bin/csh

#PBS -d workdir
#PBS -q @dg02.cc.kek.jp
cd $HOME/workdir
./test.sh
```

Figure 4.2: An example of PBS script

Traditional Approach

In the traditional approach, application developers need to follow the specific commands and rules of each kind of middleware. NAREGI uses specific commands such as
"`naregi-job-submit`"
to submit a job with a Work Flow Markup Language (WFML [81]) file. The WFML file is needed to define the job description as shown in Figure 4.1. NAREGI requires a WFML formatted file to describe the attributes for a job, such as the execution file, working directory, and so on.

The other middleware, Torque requires the specific command
"`qsub`"
to submit a job with a PBS script file, as shown in Figure 4.2. The PBS script is used to define the job description as with WFML in NAREGI.

The traditional approach forces the application developers to prepare differ-

ent formats of job descriptions and to use different commands for each kind of middleware, even if the content of the job descriptions is the same. The application developer must also insure the compatibilities with all of the middleware that they are using. Also, additional efforts are required when a user's applications are deployed in other middleware infrastructures.

4.1.2 UGI Implementation

Once a UGI adaptor (including SAGA) for each kind of middleware is prepared, application developers only need to use the functional API and do not need to worry about the features of each specific middleware. Table 4.1 shows some examples that application developers can use to invoke UGI APIs for job submissions to Grid middleware.

A UGI job description has several attributes. The application developer can configure them once and reuse the job description to submit jobs for other middleware infrastructures. A sample configuration for a UGI job description is described in Section 4.1.3. Application users need only specify the job service, such as NAREGI, gLite, or Torque, etc. and modify part of the job description to switch it to another service.

Figure 4.3 shows a detailed architecture using UGI. The UGI layer is located over a SAGA layer that is located between "End users" and several kinds of computing resources. Even if a firewall exists between computing resources and higher-level components, users can use resources from all of the middleware infrastructures through UGI. Users can use resources from all middleware infrastructures through UGI. Application developers can develop their own applications without any concerns about the underlying Gird middleware. In addition this approach provides an easy mechanism such as a Web interface that end-level users can use even behind firewalls. For the practical experiment, we have deployed a host that has the pre-installed UGI, and the required software libraries.

We created SAGA adaptors for NAREGI (SNA: SAGA NAREGI Adaptor) and for Torque (STA: SAGA Torque Adaptor) that comply with version 1.0 [37] of the specification discussed in the OGF. We can access NAREGI and Torque through SAGA in the UGI layer. The gLite SAGA adaptor was not available as of May 2012, so we prepared another adaptor for UGI because it is easier to

UGI API	**Function**
`ugi.url.url()`	Specify job service (e.g. NAREGI or Torque).
`ugi.job.description.description()`	Create a job description.
`ugi.job.service.create_job(description)`	Create a job with description.
`ugi.job.job.run()`	Submit a job.

Table 4.1: Frequently invoked UGI APIs for job submissions.

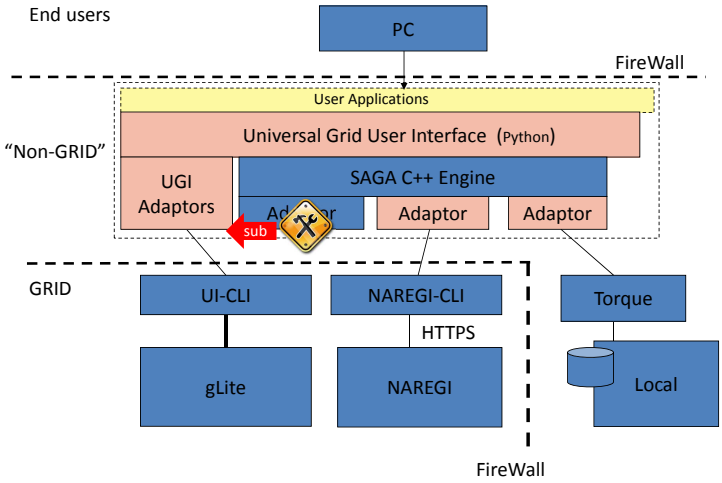


Figure 4.3: UGI-based user environment with Grid middleware.

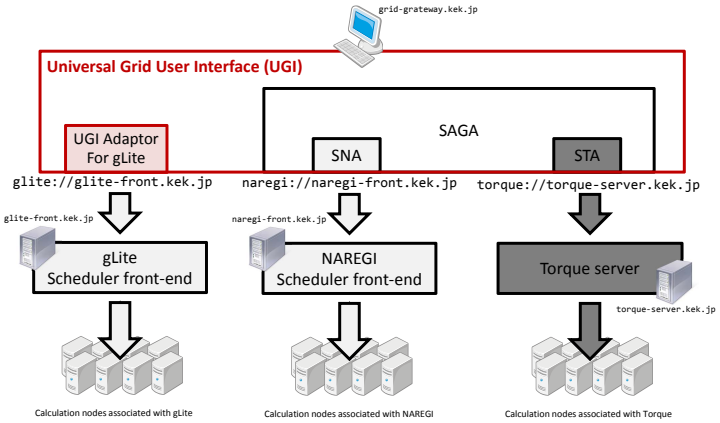


Figure 4.4: Workflow diagram in the user environment based on UGI.

implement a UGI adaptor than a SAGA adaptor. Our demonstration works in the UGI environment with the adaptors we created. UGI should be installed on a host server that is called the "UGI host". Figure 4.4 shows the workflow diagram for our demonstration.

4.1.3 Job Submission with UGI

A UGI application can be executed in NAREGI, gLite, and Torque. Figure 4.5 shows sample code (sample_ugi.py) to submit a job using the UGI API. The sample code calls another script to define a job task (Figure 4.6). In this task, the job description based on JSDL (Job Submission Description Language) [82] is simply defined in the code. The application developer can separate the job description from the code if necessary. Users need specify only a pair of a job service and a hostname as the argument to submit a job in these examples.

```
import ugi
import urlparse

argvs = sys.argv
argc = len(argvs)

middle = urlparse.urlparse(argvs[1])[0]
site = urlparse.urlparse(argvs[1])[1]
tasks = []

tk = ugi.job.task(middle)
tk.site = site
tk.share = share = [1,] # no. of jobs
tk.mode = 'multipleJob'
tk.script = script = "jobtask.py"
tasks.append(tk)

# Job submission
ugi.job.submit(tasks)
```

Figure 4.5: Job execution example using UGI.

```
import sys, saga

argvs = sys.argv
argc = len(argvs)

site = sys.argv[1]
middle = sys.argv[3]

try:
    # Create a Job Description
    url = middle + "://" + site
    js_url = saga.url(url)
    job_caht = site
    job_service = saga.job.service(js_url)
    job_desc = saga.job.description()
    job_desc.executable = './test.sh'
    job_desc.working_directory = '$HOME/work_dir'
    job_desc.candidate_hosts = job_caht
    # Submit a job
    my_job = job_service.create_job(job_desc)
    my_job.run()
except saga.exception, e:
    print "SAGA Error: ", e
```

Figure 4.6: Job task example using SAGA.

For example, here is a command to submit a job to NAREGI:

```
$ python sample_ugi.py naregi://naregi-front.kek.jp
```

The corresponding command to submit a job to Torque is:

```
$ python sample_ugi.py torque://torque-server.kek.jp
```

In the case of gLite, no SAGA adaptor is available, so UGI cannot call jobtask.py directly. Instead, UGI calls the UGI gLite adaptor. Then we can also submit a job to gLite as:

```
$ python sample_ugi.py glite://glite-server.kek.jp
```

There is no need to change the application itself as shown in this example. Application developers do not need to deal with the incompatibilities between the different kinds of middleware.

4.1.4 Demonstration Results

Our actual HENP applications created in a practical user environment with SAGA were successfully submitted to RENKEI resources on the deployed NAREGI system and also used local resources managed by Torque. We deployed a PTSim program based on Geant4 [83, 84, 85] as a real application using resources from both kinds of middleware. The application is a Monte Carlo simulation of the particle interaction of a proton beam with the materials making up a human body. Further discussion of PTSim appears in Section 6.2.

The Python script program successfully controlled the job submissions and monitored the job status. The output files of the simulation were transferred to the client host (UGI host) and post-processed to display dose distributions and particle trajectories. The whole process of this workflow was described in a simple Python program that is easy for the end users to understand. For application developers, this provides a convenient environment for debugging and tuning the application. Users can change the application parameters and submit jobs to use the resources under both kinds of middleware for rapid and easy debugging.

This demonstration showed the usability of the universal Grid interoperable environment. This also makes it possible for non-Grid applications that have always used local resources to become portable for export to distributed Grid resources.

4.2 File Manipulation in Multi-Data Grids

This section describes practical file access applications for distributed storage resources over multi-file-system middleware. At the CRC in KEK, many data

files are produced by physical experiments. Sharing bubble chamber image (Figure 4.7) files between KEK and Kings College in UK is one of the examples. We tried to display one bubble chamber image file that is divided and stored on the different kinds of file-system middleware. We used bubble chamber images that can be interpreted visually for simplicity. For larger data files, our approach will work even more efficiently.

The kinds of file-system middleware we used were iRODS and Gfarm. To access these kinds of middleware, we created the UGI file APIs and the SAGA adaptors for iRODS and Gfarm. We present the technical details for the user environment and discuss the usability by using real bubble chamber image files.

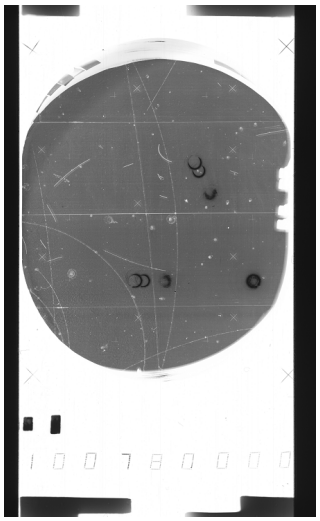


Figure 4.7: Bubble chamber photo image.

4.2.1 UGI Implementation

There are several kinds of file-system middleware in use today. Using file-system middleware requires using specific commands and rules to access files, and the differences cause problems for application developers. UGI masks the differences between different middleware infrastructures.

We use two types of file-system middleware: iRODS and Gfarm. Our demonstration shows how easy it is to access the different types of file systems. Table 4.2 shows examples of application developers using the UGI API to access the file systems. Application users need only specify the middleware scheme (e.g. iRODS or Gfarm) and each logical path to switch to other kinds of file-system middleware.

Figure 4.8 shows the detailed architecture of the Data Grid environments when using UGI. The UGI layer is located between the "End users" and the various kinds of file-system middleware. A local file-system is located in the same layer as "GRID" for our study, although the local file system is not a kind of Grid

UGI API	Function
`ugi.file.read()`	Read the file.
`ugi.file.write(*buffer)`	Write *buffer* data to the file.
`ugi.file.get_size()`	Get file size of the file.
`ugi.file.copy()`	File copy within a protocol.
`ugi.file.transfer_copy()`	Tramsfer a file across different protocol.

Table 4.2: Frequently invoked UGI APIs for file manipulations.

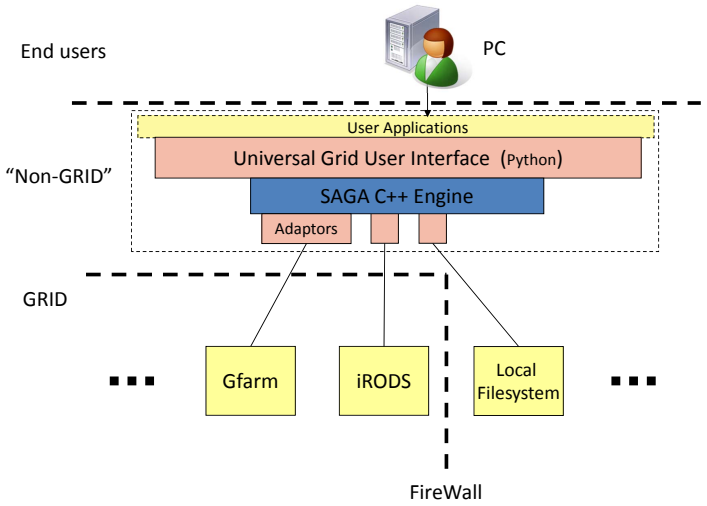


Figure 4.8: UGI-based user environment with Data Grids.

technology. That is because our goal is to use resources on different kinds of Grid middleware as well as local resources. Even if a firewall exists between the other kinds of file-system middleware and the clients, end users can access the storage resources on all of the file-system middleware, iRODS, Gfarm, and the local file system through UGI.

We created SAGA adaptors for iRODS (SIA: SAGA iRODS Adaptor) and for Gfarm (SGFA: SAGA Gfarm Adaptor). Our demonstration works in the UGI environment with the adaptors. The client applications for iRODS and Gfarm should be installed on the UGI host. In our experiments, the i-command software for iRODS and the Gfarm client software are installed on the UGI host. Figure 4.9 shows the workflow for our prototype. The command
`ugi.file.transfer_copy()`
transfers files among the different kinds of Data Grid middleware.

4.2.2 File Access with UGI

This section describes how to access files in different kinds of file-system middleware. First, we tried to share the bubble chamber photo image files among the different kinds of file-system middleware. In the next step, we divided an image

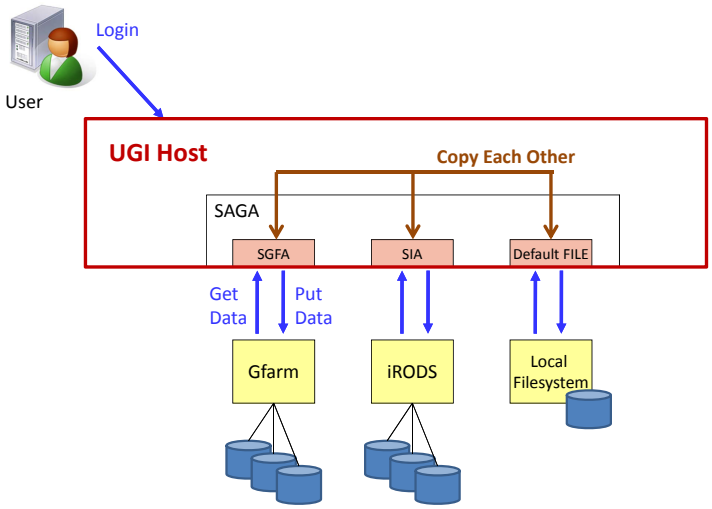


Figure 4.9: Workflow diagram in the user environment based on UGI.

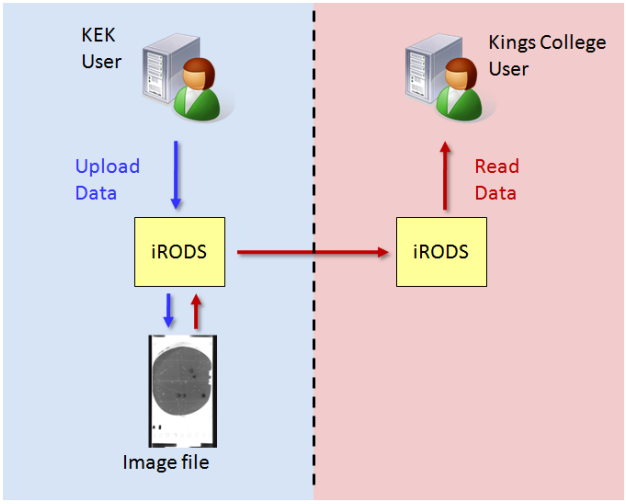


Figure 4.10: iRODS network between KEK and Kings College.

file into smaller data files and stored the pieces in the different kinds of file-system middleware. Then, we tried to combine the divided data to display the image file.

Access Files Stored in Different file systems

Figure 4.7 shows an example of a bubble chamber image. Such images are stored in iRODS during normal use. The clients in KEK and Kings College can share the image data as shown in the Figure 4.10.

We have other storage resources in another file system, Gfarm. Attaching the existing Gfarm file system to our environment is simple. This allows us to share the existing data in the Gfarm and also easily expand the storage resources. Figure 4.11 shows the clients using UGI to store and share the image files among iRODS,

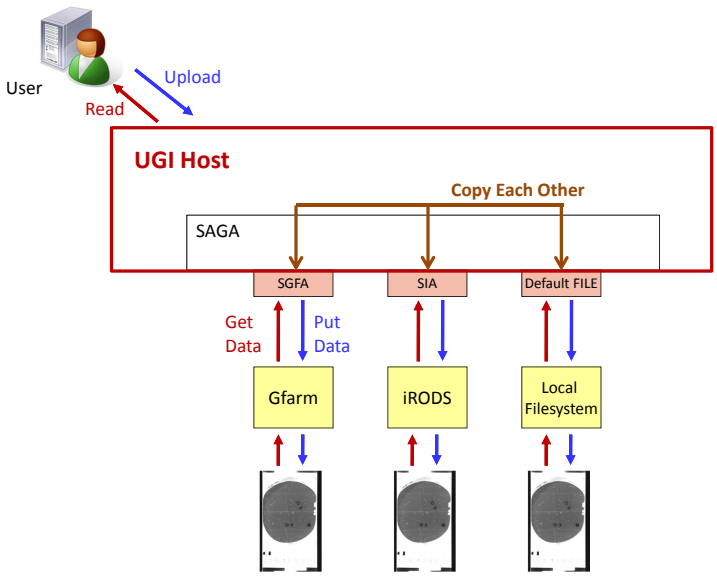


Figure 4.11: File Access via UGI.

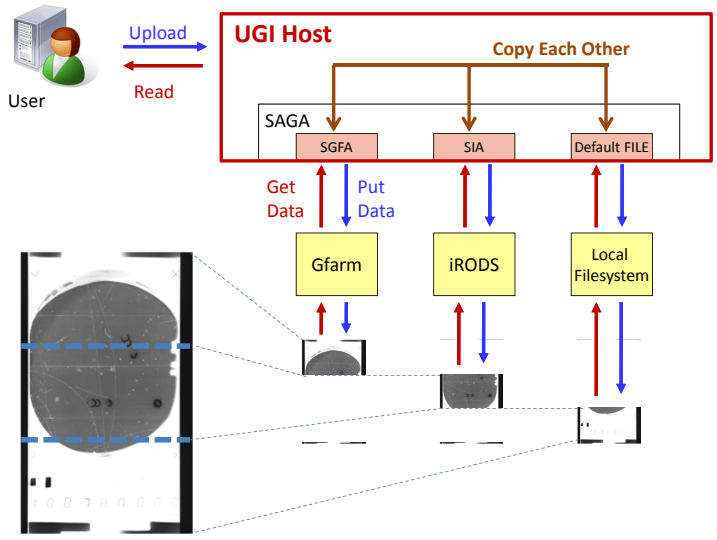


Figure 4.12: File Access to separated image data via UGI.

Gfarm, and local file systems.

Combine Distributed Data

In considering the use of large files in the next step, sharing an image file that is divided and stored on the different kinds of file-system middleware is convenient. In our tests, we divided a bubble chamber photo into three pieces and stored them on the different kinds of file-system middleware: iRODS, Gfarm, and the local file system (Figure 4.12).

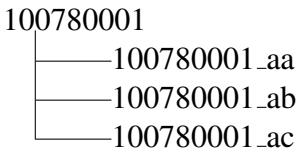
```
import ugi
import sys

argvs = sys.argv
argc = len(argvs)

img_urls = []

for line in open(argv[1], 'r'):
    img_urls.append(line)

for img_url in img_urls:
    f_size = ugi.file.get_size(img_url)
    mem = ugi.file.read (img_url, f_size)
    print mem
```

Figure 4.13: img_cat_ugi.py:The sample UGI application.

```
gfarm://localhost/kawai/bubble/100780001_aa
irods://localhost/tempZone/home/kawai/bubble/100780001_ab
file://localhost/home/kawai/bubble/100780001_ac
```

Figure 4.14: filelist.txt:The URL list of file locations.

We created a sample UGI application, the "img_cat.py" script showed in Figure 4.13, which concatenates the distributed files. The img_cat command requires the file list indicating the locations of the divided data as its argument. The Figure 4.14 shows the list of the distributed file locations. In this case, the example photo (file name is 100780001) is divided as below:



Each location is specified in the format of a URL compliant with RFC 1630 [86]. Then, we uses the "display" command of the ImageMagick software [87] to display the outputs of the img_cat command, as:

```
$ python img_cat.py filelist.txt | display
```

The UGI application combines the distributed files and displays them as an image file. There is no need to change the application if we use different kinds of file-system middleware. All we need is to change the list of file locations. Application developers do not need to worry about compatibility among the various kinds of file-system middleware.

4.2.3 Demonstration Results

Our sample application accesses the distributed files on the different kinds of file-system middleware and displays the image with the ImageMagick software. The simple program is easy for the end users to read because the coding method is similar to a traditional program accessing a local file system. Also, the users can easily change the file locations for the different kinds of file-system middleware without modifying the source code. For application developers, this provides a convenient environment for debugging and tuning their applications because they do not need to worry about the file-system differences.

In this demonstration, we displayed a bubble chamber image that is divided and stored in different kinds of file-system middleware: iRODS, Gfarm, and a local file-system. The sharing of files between KEK and Kings College will be used not only for display, but also for analyses, simulations, and modifications in the future. Using the ImageMagick API [88] or another CAD software API will be required for future applications. We have not yet studied the memory management and speed of our prototype. The simple program we showed only reads the divided data sequentially. Techniques to allocate memory and to implement multi-threading methods are also required to improve the applications.

4.3 Metadata Control in Multi-Grid Environments

This section describes the management of files distributed in heterogeneous Data Grids by using RNS. RNS provides hierarchical namespace management for name-to-resource mapping to use Grid resources for different kinds of middleware.

RNS directory entries and junction entries can contain XML messages as metadata. We define attribute expressions in XML for the RNS entries and give an algorithm to access distributed files stored within different kinds of Data Grids. The example in this section shows how our Grid application can retrieve the actual locations of files from the RNS server. An application can also access the distributed files as though they were files in the local file system without worrying about the underlying Data Grids.

4.3.1 How to Access Distributed Files

This section shows how to access distributed files in various Data Grids. As our example of distributed file access, we assume that the large data$A$ file consists of several pieces stored in different kinds of Data Grids. To manipulate data$A$, application users need to retrieve all of the distributed pieces by using a Grid application.

A Grid application needs the physical locations of the existing resources in the Data Grids. The required additional information can be attached to each RNS entry because the RNS entries can include XML metadata. We define an XML

```
<?xml version="1.0" encoding="UTF−8"?>
<file xmlns="http://kek.jp/rns/test">
  <rnskv key="file_num" xmlns="">n</rnskv>
  <rnskv key="file_data_0" xmlns="">dataA_0</rnskv>
  <rnskv key="file_data_1" xmlns="">dataA_1</rnskv>
  <rnskv key="file_data_2" xmlns="">dataA_2</rnskv>

                  :
                  :

  <rnskv key="file_data_n" xmlns="">dataA_n</rnskv>
</file>
```

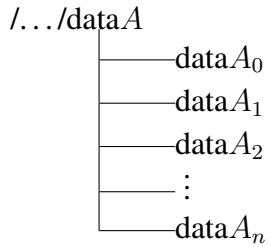
Figure 4.15: Attribute definition of virtual directory

attribute expression for each RNS entry to handle all of the physical locations. We also describe the algorithm to retrieve the information of physical locations for the distributed files.

Attribute Definition

We need to consider what kind of XML message should be contained in each RNS entry. We primarily define how to express the attribute message in the virtual directory. There are several forms for these definitions, so the last part of this subsection considers some other possibilities that would have been less suitable for managing this kind of distributed file.

RNS Virtual Directory If the data$A$ file consists of dataA_0, dataA_1, dataA_2, ..., dataA_n, the RNS path will be defined:



The file locations of the data$A$ pieces should be addressed in XML as metadata in the virtual directory entry for data$A$. Figure 4.15 shows the attribute definition of the virtual directory in XML. The <rnskv> tag contains key-value metadata as attributes. The "file_num" key indicates that this virtual directory entry represents a file consisting of several divided pieces and its value is the number of the pieces. The value of the "file_data_*i*" key should be the relative RNS junction path-name to the current RNS virtual directory entry (for data$A$).

RNS Junction Each RNS junction should be registered to refer to the physical file locations of the data$A$ pieces. Any RNS path name can be registered logically, but it is better to register the pieces directly in the data$A$ virtual directory. In particular, we must avoid specifying an absolute RNS path in the metadata for any RNS entry because an absolute RNS path can be different for each client[29].

The EPR of each junction is by itself sufficient to address an existing resource. An additional XML message can be attached to each junction when the addressed resource is replicated.

Comparison with Other Expressions

One of the other ways to address a physical file location is by directly referencing an existing resource as the location value in the "file_data_*i*" key instead of referencing an RNS path. This method can avoid registering some RNS junctions and thus improve the execution times by avoiding the processing required to access those RNS junctions. However, this method has the serious disadvantage that the XML attribute message must be modified whenever the physical location of any piece of the file changes. Also, the XML still needs to handle the other physical locations of the replicated pieces. This increases the complexity of the implementation.

Another method is to define the attribute messages in the RNS junctions. This method seems to work in the same way as with our definitions. However, this approach requires a dummy EPR to register each junction. That is because the RNS specification requires that all of the compliant RNS implementations must embody the target information of a namespace junction within a valid WSAddressing EPR[29]. Such a dummy EPR may cause side effects such as hard-to-detect address conflicts or infinite RNS path loops.

Using an RNS lookup operation is effective for obtaining all of the entries within a given target RNS virtual directory[29]. However, in this case RNS cannot control the order of the junction entries, so the RNS junction names should include ordering information. Also, if the obtained entries include invalid junctions or virtual directories, we will need additional checks to detect them.

Algorithm

The physical location list of the existing resources can be obtained by using Algorithm4.1. In this algorithm, $\mathcal{L}$ is the set containing the RNS locations specified in the XML metadata of the RNS virtual directory. The set $\mathcal{R}$ has all of the RNS junctions in the RNS namespace. The function U maps each RNS junction to an existing physical URL.

We need to add a NULL record to the $urlList$ when the corresponding physical URL does not exist. This is because the Grid application must know about the availability of each piece of the file to reconstruct data$A$.

Algorithm 4.1 Get the location list of existing resources

1: **for** each $rnsLocation\ l_i$ in $\mathcal{L}$ **do**
2: **if** $l_i \in \mathcal{R}$ **then**
3: add URL $u_i = U(l_i)$ to $urlList$
4: **else**
5: add NULL to $urlList$
6: **end if**
7: **end for**
8: return $urlList$

UGI API for metadata	Function
`ugi.file.transfer_register()`	Transfer a file to a destination and register to RNS catalog.
`uig.file.replicate()`	Replicate a file registered in RNS.
`ugi.file.set_metadata()`	Set metadata to a file registered in RNS catalog.
`ugi.file.get_metadata()`	Get metadata of a file registered in RNS catalog.
`ugi.file.get_epr()`	Get EPR of a file registered in RNS catalog.
`ugi.file.mod_metadata()`	Modify metadata of a file registered in RNS catalog.
`ugi.file.query_metadata()`	Query metadata of a file registered in RNS catalog.

Table 4.3: Frequently invoked UGI APIs to handle metadata.

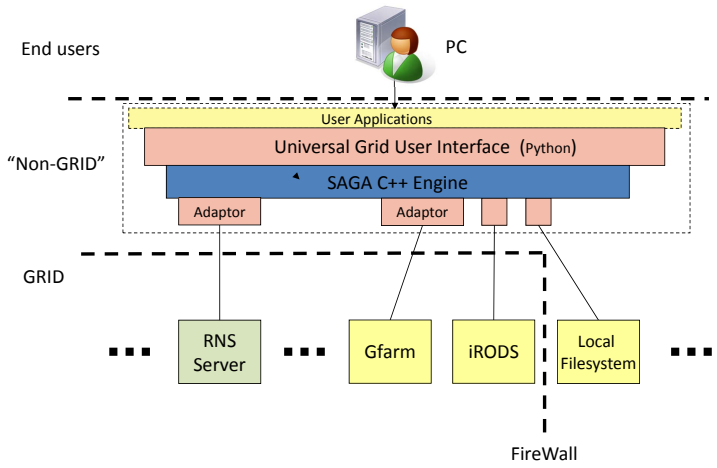


Figure 4.16: UGI-based user environment with RNS.

4.3.2 UGI Implementation

Using Data Grids involves using their specialized commands and rules to access their files. The differences among these commands and rules cause problems for application developers. UGI provides a unified interface that conceals the differences among the different middleware infrastructures. We need to control the RNS application via UGI.

Table 4.3 shows some examples of application developers calling the UGI APIs to handle metadata. Figure 4.16 shows details of the architecture using different Data Grids and RNS with UGI.

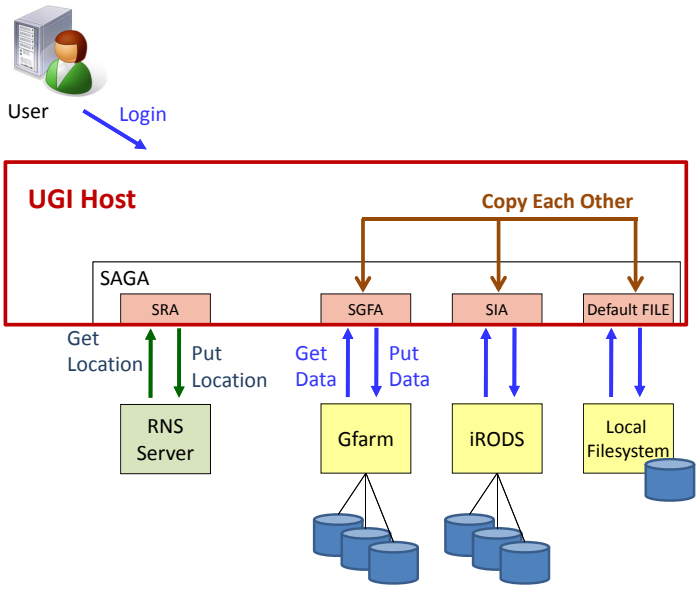


Figure 4.17: Workflow diagram in the user environment based on UGI.

Our demonstration works in a UGI environment with iRODS, Gfarm, a local file system, and RNS. UGI is prepared on the "UGI Host". Figure 4.17 shows the workflow for our experiments.

The client application should get the information about the physical file locations from an RNS server via UGI before accessing the Data Grids. When the application stores files on the Data Grids, the application should send the information about the file locations to the RNS server after storing the files.

4.3.3 UGI Example

We divided a typical bubble chamber photograph into three pieces and stored the pieces on the different Data Grids: iRODS, Gfarm, and the local file system (Figure 4.18). Also, we registered each physical file location in RNS catalog system.

We created a sample UGI application, the "img_cat_ugi.py" script shown in Figure 4.19, which retrieves the physical locations of the distributed files and concatenates the identified files. We registered an RNS virtual directory and three junctions:

```
rns://sg01.cc.kek.jp/100780001
                      ├──100780001_aa
                      ├──100780001_ab
                      └──100780001_ac
```

The RNS path-name of the virtual directory is required as an argument for the img_cat_ugi.py script. The hostname "sg01.cc.kek.jp" is the RNS server in our environment. Therefore, the RNS path-name of the argument becomes "rns://sg01.cc.kek.jp/100780001".

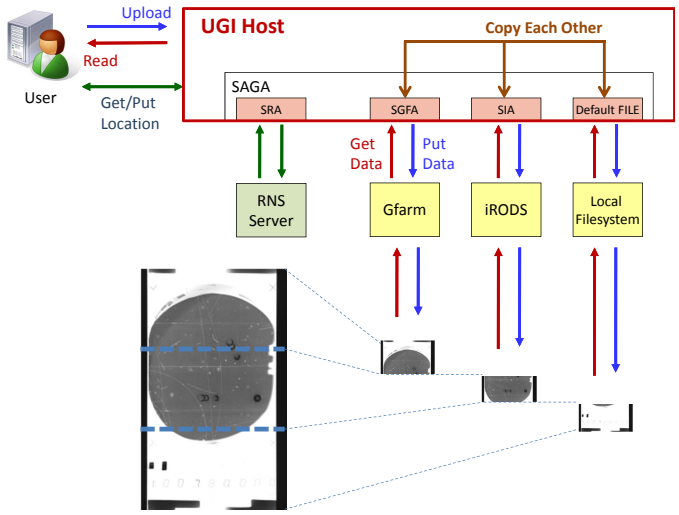


Figure 4.18: File access to separate pieces of a photograph via UGI.

```
import ugi
import sys

argvs = sys.argv
argc = len(argvs)

// get RNS junctions
u = argv[1]
str_file_num = ugi.file.get_metadata(u, "−k=file_num")
file_num = int(str_file_num)
rns_jncs = []
for i in file_num:
   rns_jncs.append("file_data_" + str(i))

// get physical locations
img_urls = []
for rns_jnc in rns_jncs:
   f_url = u + "/" + rns_jnc
   epr = ugi.file.get_epr(f_url)
   img_urls.appned(epr)

// read existing files
for img_url in img_urls:
    f_size = ugi.file.get_size(img_url)
    mem = ugi.file.read (img_url, f_size)
    print mem
```

Figure 4.19: img_cat_ugi.py – The sample UGI application with RNS.

We can specify the metadata of the RNS virtual directory, 100780001, as shown in Figure 4.20. Table 4.4 shows the physical resource locations of the divided files. Each URL for a physical location is specified in the format of a

```
<?xml version="1.0" encoding="UTF-8"?>
<file xmlns="http://kek.jp/rns/test">
  <rnskv key="file_num" xmlns="">3</rnskv>
  <rnskv key="file_data_0" xmlns="">100780001_aa</rnskv>
  <rnskv key="file_data_1" xmlns="">100780001_ab</rnskv>
  <rnskv key="file_data_2" xmlns="">100780001_ac</rnskv>
</file>
```

Figure 4.20: The example of the attribute definition

```
<ns1:EndpointReferenceType
 xsi:type="ns1:EndpointReferenceType"
 xmlns:ns1="http://www.w3.org/2005/08/addressing"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ns1:Address xsi:type="ns1:AttributedURI">
    gfarm://sg07.cc.kek.jp/kawai/bubble/100780001_ab
  </ns1:Address>
</ns1:EndpointReferenceType>
```

Gfarm Path Name

Figure 4.21: EPR example indicating Gfarm resource

File name	File System	URL
100780001_aa	iRODS	irods://sg03.cc.kek.jp/tempZone/home/kawai/bubble/100780001_aa
100780001_ab	Gfarm	gfarm://sg07.cc.kek.jp/kawai/bubble/100780001_ab
100780001_ac	Local File System	file://sg01.cc.kek.jp/home/kawai/bubble/100780001_ac

Table 4.4: Physical resource locations of the divided example files.

URL compliant with RFC1630. As an EPR example, Figure 4.21 shows the RNS junction EPR of a Gfarm file resource. The value of <ns1:Address> indicates the URL of the physical Gfarm location. For iRODS and the local file system, each URL is specified as an <ns1:Address> value for each EPR.

Then we can use the "display" command of ImageMagick to display the output of the img_cat_ugi.py script as:

```
$ python img_cat_ugi.py rns://sg01.cc.kek.jp/100780001
  | display
```

Our application retrieves the locations of the distributed files from the RNS server, combines them and displays them as an image. There is no need to change the application if we use different kinds of Data Grids. All we need to do is change the RNS entries and the metadata. Application developers do not need to worry about any incompatibilities among the different kinds of middleware.

4.3.4 Demonstration Results

The concept of managing the distributed files by RNS was validated by our prototypes. Our sample application successfully retrieves the location information for

the distributed files on the different kinds of Data Grids by using RNS and accesses the files. The application also displays them as an image with ImageMagick. This simple program is easily understood by the end users because the coding method is similar to a traditional program accessing a local file system and it only requires some simple key-value operations. The file location information is fully managed in the RNS system without modifying the source code of the application.

As an example, we displayed a bubble chamber image that was divided and stored in the different kinds of Data Grids: iRODS, Gfarm, and the local file system. We can consider several ways to store the physical locations of the divided pieces. Using RNS metadata to manage the information about the file locations supports sharing the metadata among different users. Once the RNS entries and the metadata have been created, it is easy for Grid users to access the distributed pieces without worrying about each physical file location. Sharing the metadata in RNS is beneficial because the required information associated with each file in the future work can be referred to and modified by distributed researchers all over the world.

Chapter 5

Abstraction Layer Evaluation

As we mentioned in the previous chapters, we have already prepared the necessary components for the software-abstraction layer to utilize different kinds of Grid middleware, and have described the implementations and experimental results. We must consider the overhead of the abstraction layer.

In this chapter, we evaluate the performance with a software-abstraction layer such as SAGA and consider appropriate evaluation metrics for such abstraction layers.

5.1 Overhead Evaluation

5.1.1 Inside of Abstraction Layer

We chose Torque [89] to evaluate the overhead of the abstraction layer. SAGA and the STA can be used as an abstraction layer with Torque. Figure 5.1 shows the application execution flow using STA. The job submission to the Torque system is executed by using a qsub [90] command, which is one of the Torque client commands. STA includes the qsub command internally and the qsub command is invoked by the process Boost.Process [91]. SAGA has Python bindings, so we can use the SAGA Python API.

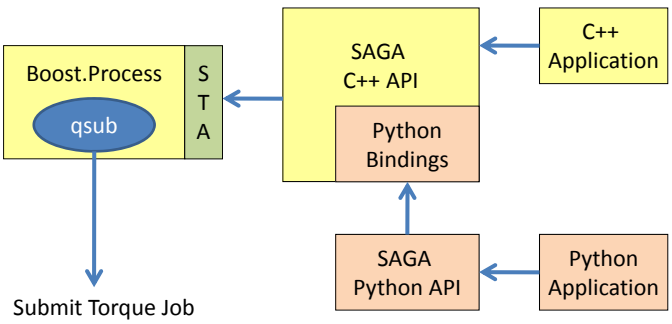


Figure 5.1: Call mechanism in SAGA with STA

```
#! /bin/sh
#PBS −N saga−app
#PBS −d /home/ykawai/tmp
#PBS −l walltime=300
#PBS −q @dg02.cc.kek.jp
#PBS −M yutaka.kawai@kek.jp
/bin/hostname
```

Figure 5.2: PBS script for overhead evaluation

Case	Average	Standard Deviation	Variation Coefficient
(A) qsub	80.17 ms	108.01	1.35
(B) C++	472.16 ms	148.76	0.32
(C) Python	539.10 ms	165.03	0.31

Table 5.1: Average and standard deviation of the performance results.

According to this design within the SAGA C++ implementation, it is clear that the performance costs are increased in an order corresponding to these three cases when submitting a job (which means that Python has the largest overhead):

(A) Direct use of qsub

(B) Using the SAGA C++ API

(C) Using the Python API

5.1.2 Evaluation for Job Submission

Performance Verification

For fair comparisons, we prepared each job description so that the content of each job is the same in the three cases. The case A uses the PBS script shown in Figure 5.2. The case B with the SAGA application written in C++ (Figure 5.3) and the case B with the Python script is shown in Figure 5.4. We used very simple jobs that only involved repeatedly obtaining the result of a "hostname" command, so the calculations for the job load were not related to our calculations of the SAGA overhead. The time test uses a "time" command in Linux and the shell script shown in Figure 5.5. All of the three cases were tested simultaneously. Each trial submitted five hundred sequential jobs.

Figure 5.6 shows the results and Table 5.1 shows the average elapsed time and standard deviation of each set of results.

```
#include <saga/saga.hpp>
#include <iostream>
namespace sja = saga::job::attributes;
namespace sjad = saga::job::attributes::detail;

int main(int argc, char* argv[])
{
  saga::url rm_url ("torque://sg01.cc.kek.jp");
  saga::job::service js(rm_url);

  saga::job::description* jd;
  jd = new saga::job::description();
  jd->set_attribute(sja::description_executable,
                    "/bin/hostname");

  std::vector <std::string> args;
  args.push_back("");
  jd->set_vector_attribute(sja::description_arguments, args);
  jd->set_attribute(sja::description_output, "");
  jd->set_attribute(sja::description_error, "");
  jd->set_attribute(sja::description_working_directory, "tmp");
  jd->set_attribute(sja::description_wall_time_limit, "300");
  std::vector <std::string> hosts;
  hosts.push_back ("dg02.cc.kek.jp");
  jd->set_vector_attribute(sja::description_candidate_hosts, hosts);

  saga::job::job j = js.create_job(*jd);

  try {
    j.run();
  } catch (saga::exception e) {
    std::cout << " [what]=" << e.what() << std::endl;
    std::cout << "[error]=" << e.get_error() << std::endl;
    return 1;
  }

  return 0;
}
```

Figure 5.3: C++ code for overhead evaluation

Discussion and Results

In Figure 5.6, the required overhead to submit a job is clearly significant. The overhead varied from 4.9 to 5.7 times of the base case. However, the overhead costs did not accumulate during continuous job execution. In other words, the average elapsed time in Table 5.1 would be the delay time perceived by the end users. The users can tolerate the results with 0.5 seconds for job submissions in real-world situations while performing their job manipulations.

The variation coefficient values for C++ and Python are almost the same ($0.31 \sim 0.32$) and both graphs appear to have similar distributions. It is reason-

```
#!/usr/local/python/bin/python
import pdb
import saga

job_serv = ”torque”
job_exec = ’/bin/hostname’
job_args = (’’,)
job_caht = (’dg02.cc.kek.jp’,)
work_dir = ’/home/ykawai/tmp/’

try:
    js_url = saga.url(job_serv + ’://sg01.cc.kek.jp/’)
    job_service = saga.job.service(js_url)
    job_desc = saga.job.description()
    job_desc.executable = job_exec
    job_desc.working_directory = work_dir
    job_desc.wall_time_limit = ’300’
    job_desc.arguments = job_args
    job_desc.candidate_hosts = job_caht

    my_job = job_service.create_job(job_desc)
    my_job.run()

except saga.exception, e:
  print ”SAGA Error: ”, e
```

Figure 5.4: Python script for overhead evaluation

```
#!/bin/sh

for i in ‘seq 1 500‘
do
        (time ./jobtest) 2>> time_c.log
        (time ./jobtest.py) 2>> time_py.log
        (time qsub pbs_script.txt ) 2>> time_q.log
done
```

Figure 5.5: Shell script to execute time commands.

able to use average values to compare the cost deltas between Python and C++. This indicates the Python cost is 14.2% higher than the C++ cost.

5.1.3 Evaluation for File Manipulation

In this section we describe the tests carried out to assess the impact of the overhead of the abstraction layer while using RNS for the file manipulations. We found that the Python costs are similar to the C++ costs in the previous section, so the original commands and the C++ SAGA implementation are used here. As a baseline, the first test was run without SAGA and RNS. The second test assesses the overhead

Figure 5.6: Job submission performance in Torque

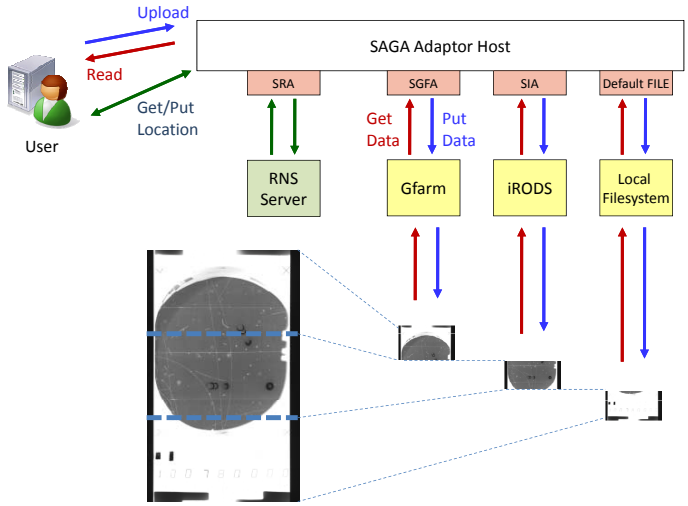


Figure 5.7: File access to separate pieces of a photograph via SAGA.

of SAGA and the last test assesses the overhead of the RNS.

Test Environment

The test environment consists of the iRODS server, the Gfarm server, the RNS server, and the SAGA Adaptor host. Those machines run on CentOS 5.5 as virtual machine guests on the same physical machine. The physical machine has an Intel CPU i7-920 at 2.67GHz with 12GB of RAM. Each guest OS runs with 1GB of RAM and one dedicated core. All of the machines are located on the same subnet. The test environment can have a non-trivial and noticeable effect on the results of such tests. Figure 5.7 shows an example of dividing and storing a bubble chamber.

Test Cases

We evaluated our implementations for the transfer of several sizes of files ranging from 100MB to 1GB. For the evaluation, we stored a third of each file on each of the three different Data Grids: iRODS, Gfarm and the local file system. Then we combined the pieces and produced a complete file in the demonstration. For example, with the 100MB file, we divided the file into 33MB pieces and stored them in the different Data Grids. In the evaluation, we then combined the 33MB pieces to create a complete 100MB file, measuring the elapsed time to combine the pieces. For average values, the test program ran three times. We compared three test configurations:

- Case 1: Normal Case
- Case 2: Case with SAGA
- Case 3: Case with SAGA and RNS

Here are detailed descriptions of the test cases:

Case 1 – Normal Case:

The normal case is a traditional situation. A list file of the file locations is contained on a local file system and the specific commands for the Data Grids are used. This case is the most basic because the list of files cannot be shared and the commands to access the Data Grids are specified in the source code. Figure 5.8 shows sample source code for the "img_cat_cmd" commands for Case 1.

Case 2 – Case with SAGA:

This case is using SAGA. A list file of the file locations is still contained on a local file system in this case, so the list file cannot be shared. However, this case uses SAGA so that we can use standardized interfaces instead of specific commands for the Data Grids. Figure 5.9 shows sample source code for the "img_cat_saga" commands for Case 2.

Case 3 – Case with SAGA and RNS:

This case is using both SAGA and RNS, which is the focus of this study. We can use standardized SAGA commands and the information about the file locations is contained in the RNS catalogue. Therefore, the file locations can also be shared. This sample source code for the "img_cat_saga_rns" commands is shown in Figure 5.10.

Test Results

Figure 5.11 shows the results of the tests. Case 2 is about 49.3% faster Case 1. That means that the cost of I/O access through the abstraction layer of SAGA is

```
#include <saga/saga.hpp>
#include <fstream>

int main (int argc, char** argv)
{ // open a list file
   std::ifstream ifs;
   ifs.open(argv[1]);

   // read the list file
   std::vector<saga::url> img_urls;
   std::string input;
   while (std::getline(ifs, input)) {
      img_urls.push_back(input); }

   // read iRODS file
   char sys_cmd[1024];
   memset(sys_cmd, '\0', sizeof(char));
   strcat(sys_cmd, "iget ");
   strcat(sys_cmd, img_urls[0].get_path().c_str());
   strcat(sys_cmd, " -;");

   // read Gfarm file
   strcat(sys_cmd, "gfexport ");
   strcat(sys_cmd, img_urls[1].get_path().c_str());
   strcat(sys_cmd, ";");

   // read local file
   strcat(sys_cmd, "cat ");
   strcat(sys_cmd, img_urls[2].get_path().c_str());

   system(sys_cmd);
   return (0);
}
```

Figure 5.8: img_cat_cmd.cpp – The sample code for Case 1.

less than the cost of using the specified commands of the Data Grids. Case 3 takes about 10 seconds longer than Case 2. The latency involves the accesses to the RNS catalogue service. The current RNS is implemented in the Java language, so the latency includes the cost of loading the Java VM.

This impact on the performance of using RNS should be considered. However, when considering the optimization of RNS implementations, it is practical to control the tradeoff between access speed and high usability. When we combine a large file, the ratio of the latency becomes smaller because the overhead of RNS is always less than 10 seconds. Actually, for 100 MB to 500 MB files, Case 3 is about 61.9% slower than Case 2. In contrast, for larger files from 500MB to 1GB, it is about 18.5% slower. Therefore, our implementation works better with files that are larger than 500MB.

```
#include <saga/saga.hpp>
#include <fstream>

int main (int argc, char** argv)
{ // open a list file
  std::ifstream ifs;
  ifs.open(argv[1]);

  // read the list file
  std::vector<saga::url> img_urls;
  std::string input;
  while (std::getline(ifs, input)) {
    img_urls.push_back(input); }

  unsigned int bs=1024*1024*5;
  char *mem = new char[bs];
  saga::mutable_buffer buf (mem, bs);

  // read existing files
  for(unsigned int i=0; i<img_urls.size(); i++){
  saga::filesystem::file f (img_urls[i]);
  const int f_size = f.get_size();
  for(unsigned int i=0; i<=(f_size/bs); i++){
    f.read (buf);
    if(i==(f_size/bs))
      std::cout << std::string (mem, (f_size%bs));
    else
      std::cout << std::string (mem, bs); } }

  return (0);
}
```

Figure 5.9: img_cat_saga.cpp – The sample code for Case 2.

Discussions

The speed performance was evaluated by three kinds of test cases. The first test case is the most basic way without SAGA and RNS. The second one looks at the impact of the overhead of SAGA. The last case concerns the overhead for SAGA and the RNS service. We found that the performance using the SAGA implementation is better than using the specific commands of the Data Grids, so this matter is discussed in Section 5.2. Certainly, the RNS implementation has some latency because of loading the Java VM and accessing the RNS service. However, the ratio of the latency decreases when combining larger files.

5.2 Evaluation Method for Abstraction Layer

There are several points to consider when evaluating such an abstraction layer. One of the points is that we need to verify is whether or not the existing software

```
#include <saga/saga.hpp>
#include <iostream>
#include <sstream>
#include <string>

int main (int argc, char** argv)
{ // Open RNS directory
  saga::url u (argv[1]);
  saga::replica::logical_directory ld (u, saga::replica::Create
                                        | saga::replica::ReadWrite);

  // get RNS junctions
  unsigned int file_num;
  std::string str_file_num = ld.get_attribute("file_num");
  std::vector<std::string> rns_jncs;
  std::stringstream ss;
  std::istringstream iss(str_file_num.data());
  iss >> file_num;
  for(unsigned int i=0; i<file_num; i++){
    ss << i;
    rns_jncs.push_back(ld.get_attribute("file_data_" + ss.str()));
    ss.str(""); }

  // get physical locations
  std::vector<saga::url> img_urls;
  for(unsigned int i=0; i<rns_jncs.size(); i++){
    saga::url f_url = u.get_string() + "/" + rns_jncs[i];
    saga::replica::logical_file lf (f_url, saga::replica::Create
                                        | saga::replica::ReadWrite);
    std::vector<saga::url> epr_list = lf.list_locations();
    img_urls.push_back(epr_list[0]); }

  unsigned int bs=1024*1024*5;
  char *mem = new char[bs];
  saga::mutable_buffer buf (mem, bs);

  // read existing files
  for(unsigned int i=0; i<img_urls.size(); i++){
    saga::filesystem::file f (img_urls[i]);
    const int f_size = f.get_size();
    for(unsigned int i=0; i<=(f_size/bs); i++){
      f.read (buf);
      if(i==(f_size/bs))
        std::cout << std::string (mem, (f_size%bs));
      else
        std::cout << std::string (mem, bs); } }
  return (0);
}
```

Figure 5.10: img_cat_saga_rns.cpp – The sample code for Case 3.

that is being considered as a candidate comparison tool is suitable for the performance evaluation. To evaluate recently developed software, the pros and cons

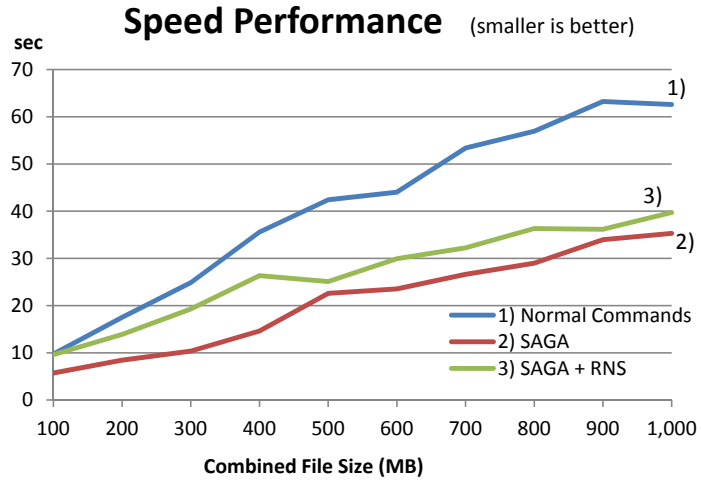


Figure 5.11: Performance results of file manipulation with SAGA and RNS

of its functions and performance must be assessed. If the selected software for the comparison is unsuitable, then it is difficult to correctly determine the performance loss of the accesses going through the abstraction layer. We discuss how to evaluate the overhead of SAGA-based applications in this section.

5.2.1 Application Example

The application example was introduced in Section 5.1.3. We tried to store each piece of the file in the different kinds of Data Grids by dividing a large file into three pieces and storing one piece in each of iRODS, Gfarm, and the local file system. Figure 5.7 shows an example of dividing and storing a bubble chamber image.

We had a sample command, "img_cat_saga" in C++, as shown in Figure 5.9 in Section 5.1.3, which concatenates the distributed files and outputs them to standard output. We use the "display" command of the ImageMagick software to display the outputs of the "img_cat_saga" command.

5.2.2 Evaluation Results

We evaluated the overhead of SAGA with the SAGA adaptors and the test application from Section 5.1. This section describes the test environment, test cases, and test results. We expected that the results would measure certain overhead costs, but the results were contrary to our expectations.

Test Environment

The test host servers consist of three servers: an iRODS server, Gfarm server, and SAGA adaptor host. The detail environments of the host servers are almost same

as Section 5.1.3 except for Gfarm authentication where we use the GSI authentication mode.

Test Case

The file size of the large file was adjusted from 100 MB to 1 GB in increments of 100 MB. As the same as Section 5.1.3, to prepare for this test, all of the files were divided in advance into three pieces and one piece was stored in iRODS, Gfarm, and the local file system. The first 1/3 of the file is stored in iRODS and the second and third pieces are stored in Gfarm and the local file system, respectively.

The method of the test is measuring the elapsed time the following actions:

1. Reading the divided pieces from the different Data Grids.
2. Outputting the concatenated pieces to standard output.

There were two test cases.

Case 1 – Traditional Case

In this case, each client command that is needed for each Data Grid system is used. This means that i-commands are used for iRODS, gf-commands are used for Gfarm, and shell commands on a Linux system are used for the local file system.

Case 2 – SAGA Case

In this case, a simple SAGA-based application is created and executed. As mentioned above, the command "img_cat_saga" is used for this test (as shown in Figure 5.9 in Section 5.1.3).

Test Results

Figure 5.12 shows the test results. The graph shows a similar trend line for all file sizes. There were no significant differences observed between the traditional and SAGA cases. Due to the architecture of an abstraction layer such as SAGA, there should be some SAGA overhead requiring more time than in the traditional case. However, the test results indicate that the SAGA overhead was negligible, which indicated that our evaluation method needed improvements.

5.2.3 Correcting Comparison Tool

We determined that the problem with this naive performance evaluation method was the use of the existing commands in the traditional case. Both the client commands for the Data Grids and the standard shell commands on Linux systems are mature in their operations and functions . In contrast, the SAGA application

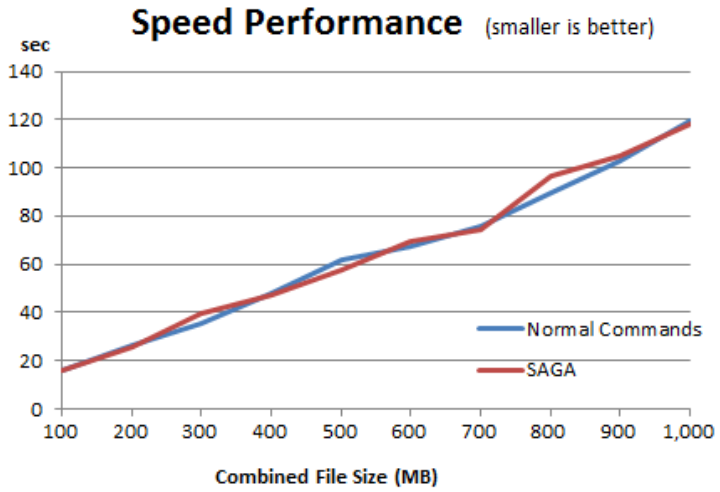


Figure 5.12: Performance results of file manipulation with normal commands and SAGA

we created with our prototype uses only the APIs of each of the client libraries, so the application is not mature and has a limited number of functions, making it is difficult to compare the new SAGA applications by simply comparisons with the existing commands.

We used three different file systems simultaneously in the previous test cases and thus, it is necessary to check whether similar results are shown with a single file system. Figure 5.13, 5.14, 5.15 show the performance results of use of each file system. The results of the iRODS and local file-system cases show that the existing commands are slower than SAGA use cases. We can say that these existing commands are not suitable to use as a comparison tool in this test.

Additionally, we found no differences between Gfarm and Gfarm with SAGA but the results of Gfarm is significantly different from (slower than) other file-system ones. That is because we used Gfarm with the GSI authentication method. Gfarm has both GSI and Shared Secret authentication methods. Figure 5.16 shows the performance differences of Gfarm between GSI and Shared Secret. The Shared Secret method is very fast, and is easy to use because of no requirement to acquire a public key. The performance curve of using the Shared Secret method in Gfarm becomes similar to that of other file systems. On the other hand, the GSI method supports data encryption and the encryption results in performance slower than the case of the Shared Secret method. We typically used Gfarm with the GSI method in terms of a variety of features for comparison.

We created a modified new iRODS command with a limited number of APIs, similar to the APIs the SAGA application includes. The results of the new performance evaluation shows that the modified iRODS command is relatively faster than the SAGA application (Figure 5.17). Therefore, using the modified new

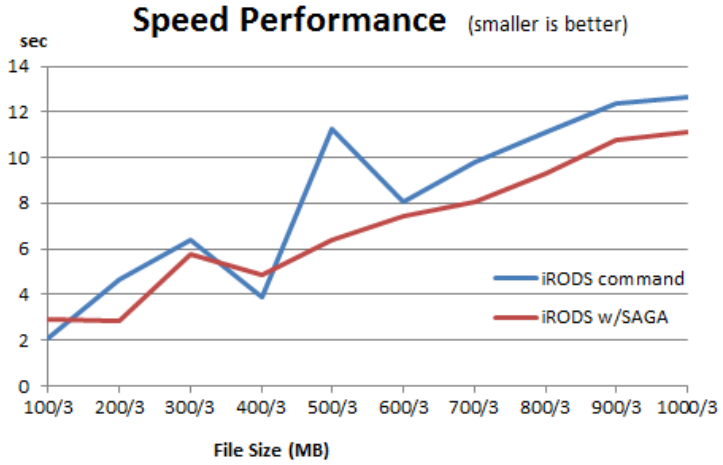


Figure 5.13: i-commands vs. SAGA and iRODS

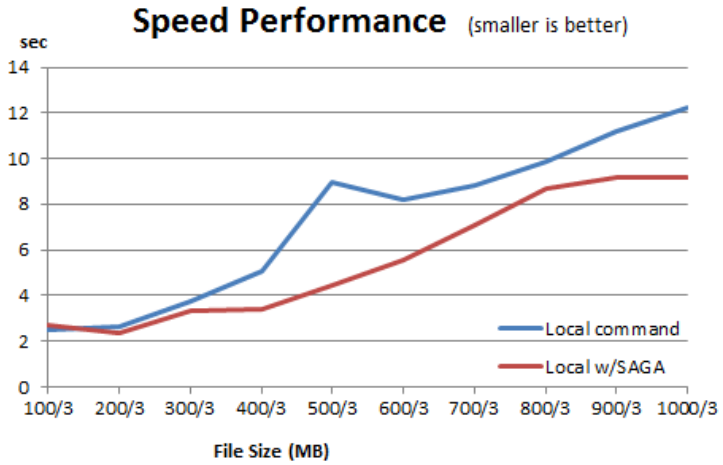


Figure 5.14: cat command vs. SAGA and local-file system

command as a comparison tool is suitable to measure the overhead of the SAGA abstraction layer. Also, we created a modified new cat command for the local file system in the same way. In this case, using the new cat command is faster than the SAGA application (Figure 5.18).

Therefore, we must pay careful attention to the maturity of the comparison tools if the tools are already in the current systems. Whenever using existing commands or APIs as a comparison tool, it is necessary to investigate inside of the existing commands or APIs. If the existing commands or APIs have some extra functions that are not related to the evaluation, we need to remove the extra functions in the commands and APIs.

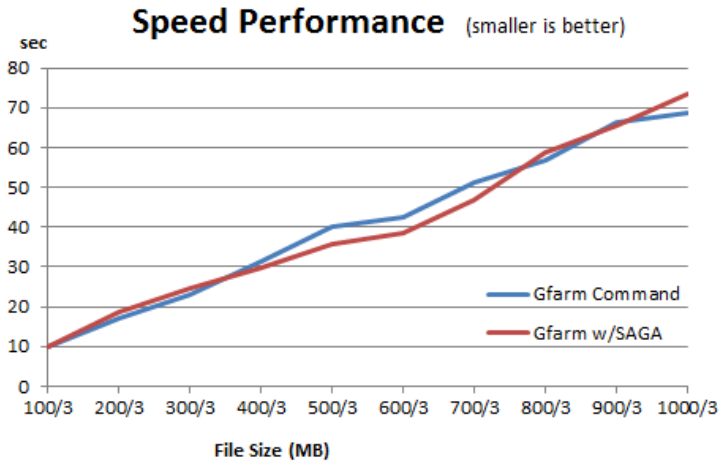


Figure 5.15: gf-commands vs. SAGA and Gfarm

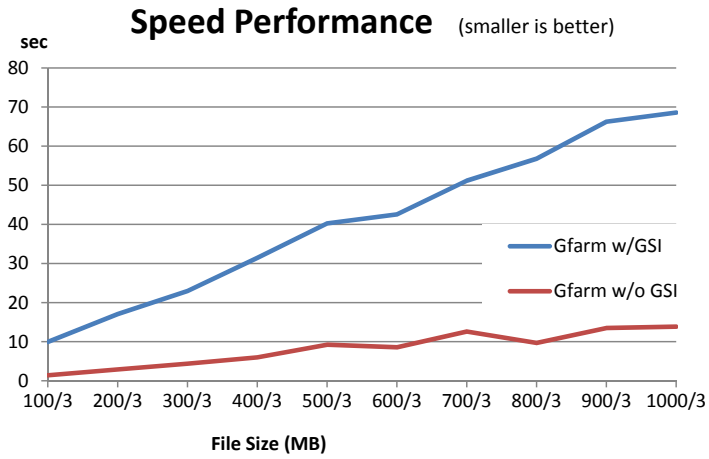


Figure 5.16: Gfarm: GSI vs. Shared Secret

5.2.4 Discussion

The results of the performance evaluation for software-abstraction layer can be fluctuated by the definition of the comparison tool. Using the existing commands and API is one of options for the evaluation. However, it is possible that we encounter some difficulties to discuss overhead costs, functionalities or tradeoff matters if we ignore the investigation of the existing commands and APIs. In other words, if we can correctly investigate the existing commands and APIs, we can prepare suitable comparison tools with some modifications.

In a pre-production stage, it is better to avoid the situation that we cannot correctly assess the overheads by simply using the existing commands and APIs as a comparison tool. That is because the prototype software adds extra functions up to the production release. For the software-abstraction layer, verifying its overhead

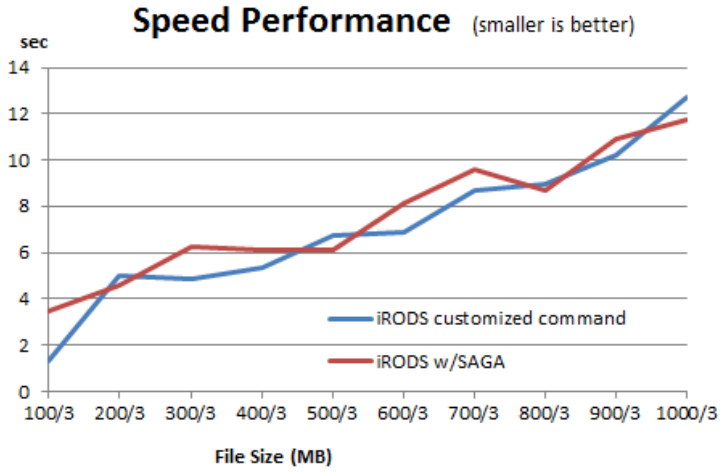


Figure 5.17: customized i-commands vs. SAGA and iRODS

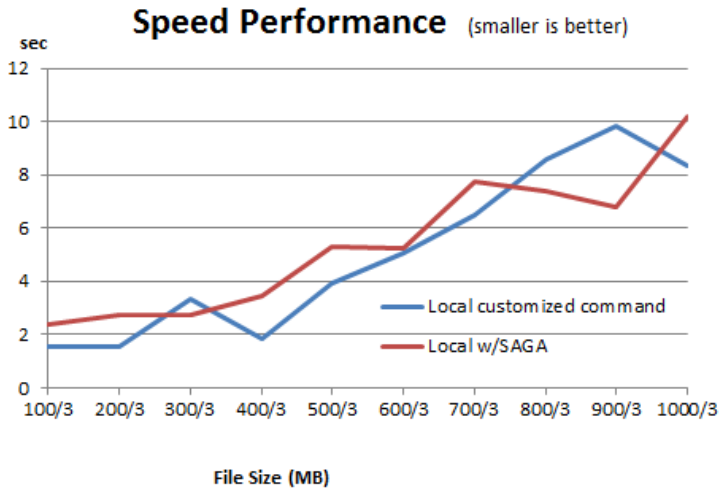


Figure 5.18: customized cat vs. SAGA and local-file system

cost is one of the requirements. It is necessary to use a correct comparison tool and to figure out the correct costs of the abstraction layer.

Chapter 6

Example Tools and Applications

6.1 Reliably Managing Files with RNS

This section describes a method for reliably managing files distributed in different kinds of Data Grids with RNS . RNS provides hierarchical namespace management for name-to-resource mapping when using Grid resources for different kinds of middleware. We define attribute expressions in XML for the RNS entries and give algorithms to access distributed files stored within different kinds of Data Grids.

The volume of digital data and the size of a typical individual file are increasing due to the introduction of high-resolution images, high-definition audiovisual files, etc. The reliable storage of such large files is becoming problematic for whole file replication, since a failure in the integrity of the file may be difficult to localize. Our method involves managing large files in Data Grids by splitting them into smaller units in a traceable manner and then managing the smaller units. The RNS catalog service uses the EPR and metadata that describe the original locations as well as the checksum values. The example shows how our Grid application can retrieve the actual file locations and the checksum values from the RNS service via SAGA and UGI.

This approach can be used with various Data Grid systems to enhance file reliability.

6.1.1 Background

The volume of digital data and the sizes of the individual files are increasing due to the introduction of high-resolution images, high-definition audiovisual files, etc. The reliable storage of such large files is becoming problematic and replication failures anywhere in such a file are difficult to localize.

We describe a method of managing large files in different kinds of Data Grids by splitting them into smaller units in a traceable manner and managing the smaller units with the RNS. RNS catalog server contains the metadata that describes the original locations of the divided pieces and their MD5 checksum values. We also

describe the tools developed to demonstrate the method that allows a file to be split before ingestion into Data Grids and assembled after extraction from the Data Grids. We can store metadata information in RNS and RNS allows the distributed files to be discovered in the Grid systems.

We describe the use of metadata in RNS to manage files distributed in different kinds of Data Grids. We use two kinds of Data Grids: iRODS and Gfarm. The example involves sharing large files using an application based on SAGA to span the environment with iRODS and Gfarm. We already showed how to manage distributed files with RNS in such heterogeneous Data Grids[92] (Section 4.3). That section presented an easier way to share the metadata about each file between the sites of different research organizations for ongoing use. This section uses the information for the physical file locations and the MD5 checksum values to enhance the reliability of the files.

6.1.2 Related Work about Reliable File Management

Various projects are underway to reliably manage files and to aggregate different heterogeneous Data Grids or file systems for efficient data sharing. Here is a summary of work related to various aspects of our study.

Luis E G. Sarmenta presented sabotage-tolerance mechanisms that work without depending on checksums or cryptographic techniques[93]. A new mechanism using voting and spot-checking together with credibility-based fault-tolerance reduced the error rates. However, such a complex mechanism requires understanding the approach and implementing it within existing systems. Also, they assumed that the mechanism would be applied to only one type of storage system. It would be difficult to apply it to different kinds of Data Grids simultaneously.

Jerzy Kaczmarek et al presented the concept and architecture of the ICAR System (Integrity Checking And Restoring System)[94]. The ICAR System not only covers the functions of integrity checkers but also automatically restores files. However, ICAR is designed as a kernel module for the operating system, and hence it is mainly used for local file systems. This is not suitable for use with Data Grids.

6.1.3 Access to Distributed Files with RNS

As our example of distributed file access, we assume that the large file data$A$ consists of several file pieces stored in different kinds of Data Grids. To manipulate data$A$, application users need to retrieve all of the distributed pieces by using a Grid application.

A Grid application needs the physical locations of the existing resources in the Data Grids. The required additional information can be attached to each RNS entry because RNS entries can include XML metadata. We have defined XML attribute expressions for each RNS entry to handle all of the physical locations.

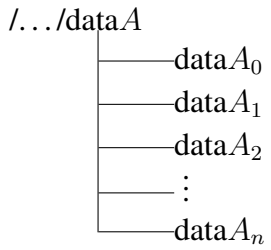
```
<?xml version="1.0" encoding="UTF−8"?>
<file xmlns="http://kek.jp/rns/test">
  <rnskv key="checksum" xmlns="">f1c9645dbc14efddc7d8a322685f26eb</rnskv>
</file>
```

Figure 6.1: Metadata example contains checksum value

An RNS virtual directory contains RNS junctions that have the physical location and the checksum value for each piece of the file. The EPR contains the physical file location and the metadata of the RNS junction contains its checksum value. The algorithms to retrieve the data from the physical location and the checksum value for the distributed files are described next.

File Piece Information

The expression of the file piece information in RNS is the same as the procedure that is already discussed in Section 4.3.1. If the data$A$ consists of dataA_0, dataA_1, dataA_2, ..., dataA_n, then the RNS path will be defined in this way:



The dataA_i RNS junctions contain the file piece information. These RNS junctions can be listed by pointing at data$A$ in the RNS virtual directory. The RNS virtual directories directly return the number of RNS junctions within themselves.

Physical Location and Checksum Value

Each RNS junction should be registered to refer to the physical file locations of the data$A$ pieces. The EPR of each junction is by itself sufficient to address an existing resource. The MD5 checksum value should be contained in each RNS junction as metadata. The EPR example for an RNS junction is already shown in Figure 4.21 in Section 4.3.3. Figure 6.1 shows metadata for an RNS junction in XML.

Algorithm

The physical location list can be obtained by using Algorithm4.1, which was described in Section 4.3.1. The checksum list ($chkList$) of the file piece can be obtained by using Algorithm6.1. The function C maps each RNS junction to a

```
 // Open RNS directory
saga::url u (argv[1]);
saga::replica::logical_directory ld (u, saga::replica::Create | saga::replica::ReadWrite);

// get RNS junctions
std::vector<saga::url> rns_jncs = ld.list();

// get physical locations and md5 checksum values
std::vector<saga::url> img_urls;
std::vector<std::string> img_md5s;
for(unsigned int i=0; i<rns_jncs.size(); i++){
        saga::url f_url = u.get_string() + "/" + rns_jncs[i].get_string();
        saga::replica::logical_file lf (f_url, saga::replica::Create | saga::replica::ReadWrite);
        std::vector<saga::url> epr_list = lf.list_locations();
        img_urls.push_back(epr_list[0]);
        img_md5s.push_back(lf.get_attribute("checksum"));
}
```

Figure 6.2: A part of SAGA C++ source example

checksum string of an existing file. Algorithm 4.1 and Algorithm 6.1 can combine the files. Part of the SAGA C++ source code that gets the physical location list and the checksum list is shown in Figure 6.2.

Algorithm 6.1 Get the checksum list of existing resources

1: **for** each $rnsJunction\ l_i$ in $\mathcal{L}$ **do**
2: **if** $l_i \in \mathcal{R}$ **then**
3: add MD5CHECKSUM $c_i = C(l_i)$ to $chkList$
4: **else**
5: add NULL to $chkList$
6: **end if**
7: **end for**
8: return $chkList$

6.1.4 Current Checksum Approach

The traditional way to checksum a large file (such as 50 GB) is by preparing the checksum value for the whole file and then reevaluate the checksum after a certain period of time[95]. Such an approach has two main problems: recomputing the checksum for the entire file is time consuming, and it is not possible to identify the exact location of a discrepancy when the checksum comparison fails. If we encounter checksum errors, there will be time-consuming work to redo the entire operation or to find a copy of the file that has retained its integrity.

A typical application involving large files involves audiovisual data. For example, television broadcasters need to preserve large archives of audiovisual data.

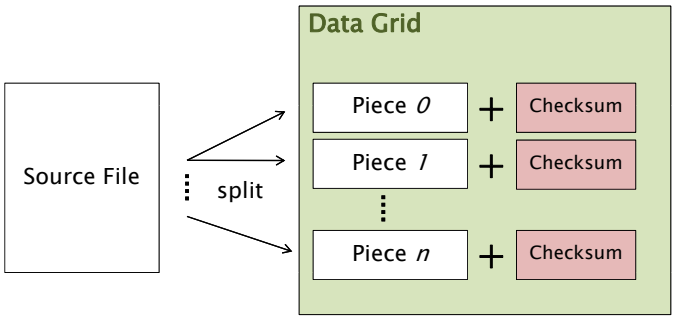


Figure 6.3: Splitting a file with checksum

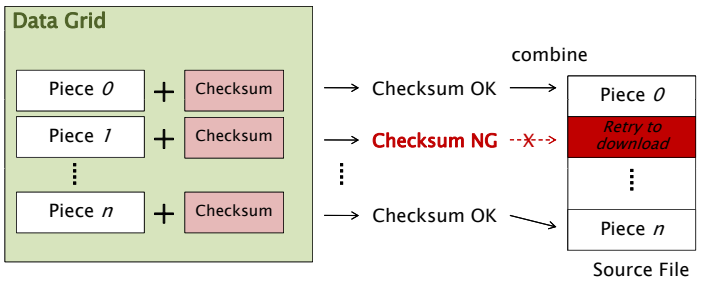


Figure 6.4: Combining pieces with comparing checksum

Movie files are very large and in some cases compressed. Using the traditional approach to checksum an entire file, it is not easy to figure out where a problem is or how to fix it if the downloaded data has incorrect or missing data.

6.1.5 Split and Checksum Approach

Our approach is to split each large file into smaller pieces, then checksum each piece and assemble the file again when it is accessed. This approach reduces the risks of incorrect or missing data. Figure 6.3 shows how to split a file and Figure 6.4 shows how to combine the pieces. If we encounter checksum discrepancies, we can identify the problematic piece and retry the download of that piece to resolve the discrepancy. This approach is much faster than the traditional method of erasing and downloading all of the data. To implement this approach, we developed a command that can split a large file into smaller pieces and store the pieces into Data Grids with their metadata. The metadata information can also be stored in RNS with XML expressions.

SAGA supports storing the pieces in different kinds of Data Grids. Figure 6.5 shows an example. If we divide a file into 100 pieces, we can store the first pieces from #0 to #33 in iRODS. The next pieces from #34 to #66 can be stored in Gfarm and the remaining pieces will be stored in the local file system. Each

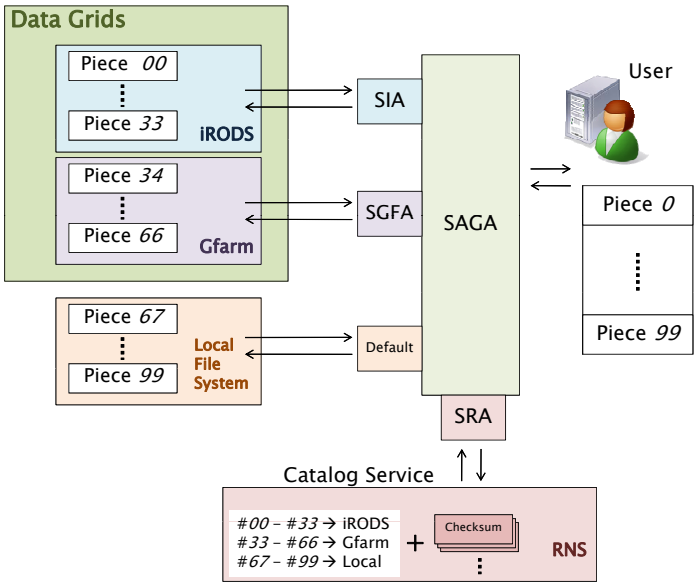


Figure 6.5: Access to distributed pieces in different Data Grids

physical location and checksum value will be recorded by the RNS catalog service via SAGA. We do not need to worry about the different interfaces of the Data Grids.

6.1.6 Performance Evaluation

In this section we describe some tests carried out to assess the overhead of checking the MD5 checksum values of the divided pieces. We used a 1GB file and divided it into different numbers of pieces from 10 to 100. We evaluated the elapsed time to download the pieces for each collection.

Also, we evaluated the overhead of SAGA and RNS. The first test was run without SAGA. The next test assessed the overhead of SAGA with RNS.

Test Environment

The test environment consists of an iRODS server, a Gfarm server, an RNS server, and a SAGA adaptor host. The detail environments of the servers are same as Section 5.1.3

Test Cases

We evaluated our implementations for the transfer of different numbers of pieces: 1, 10, and continuing to 100. For the evaluation, we stored all of the pieces on the three different Data Grids: iRODS, Gfarm, and the local file system. Then we downloaded the pieces and compared each checksum value. For example, in the case of iRODS with 20 pieces, we divided the 1-GB source file into 50-MB pieces

and stored them in the iRODS system. In the evaluation, we then downloaded all of the 50-MB pieces and compared the checksum of each piece, measuring the elapsed time to download the pieces and compare the checksum values. To obtain average values, each test program ran three times. We compared six test configurations:

- Case 1: iRODS without SAGA
- Case 2: Gfarm without SAGA
- Case 3: iRODS with SAGA and RNS
- Case 4: Gfarm with SAGA and RNS
- Case 5: Local File System with SAGA and RNS
- Case 6: Mixture of iRODS, Gfarm and the Local File System with SAGA and RNS

Here are detailed descriptions of the test cases:

Case 1, 2: Cases without SAGA The normal cases are traditional situations. The specific commands for the Data Grids are used. Each checksum value is contained in each Data Grid as its own metadata. This case is the most basic because the list of checksum values cannot be shared and the commands to access the Data Grids are specified in the source code.

Case 3, 4, 5: Cases with SAGA and RNS These cases are using SAGA with RNS. We can use standardized SAGA interfaces and the information about the file locations and checksum values are contained in the RNS catalogue. Therefore, the file locations and the checksum values can be shared.

Case 6: Mixture of Different Data Grids with SAGA and RNS The mechanism of this case is same as for Cases 3, 4, and 5. In addition, in this case, we stored a third of the pieces on each of the three different Data Grids: iRODS, Gfarm and the local file system. For example, in the case of 10 pieces, we divided the 1-GB source file into 100MB pieces and stored them in the three different Data Grids: 3 pieces in iRODS, 3 pieces in Gfarm, and 4 pieces in the local file system.

Test Results and Discussion

Figure 6.6 shows the results of the tests without SAGA. The overhead to compare the checksum values in Case 1 grows linearly. The 10 piece version of Case 1 is about 10.9% slower than the non-divided cases, which is not so different. In contrast, the 100 piece version of Case 1 is about 54.4% slower on average than the non-divided case which is unacceptably slow.

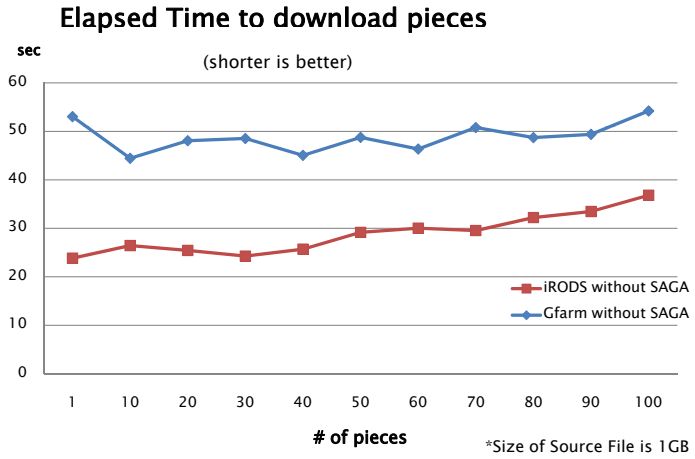


Figure 6.6: Performance evaluation results without SAGA

The latency involves accessing each piece to compare the checksum. As the number of pieces increases, the elapsed time increases. However, in the versions from 10 to 40 pieces for Case 1, the elapsed times are almost the same as for the non-divided case. In these cases the checksum values can be compared efficiently because the checksum overhead is absorbed in its access latency, which is around 25 seconds. Therefore, we can use this approach without worrying about the loss of speed when we are dividing a large file into fewer than 40 pieces for Case 1. The same conclusion applies to the 10 to 90 piece versions of Case 2. The checksum overhead of Case 2 is absorbed in its access latency, which is around 50 seconds.

Figure 6.7 shows the results of the tests with SAGA and RNS. The overhead to use the SAGA abstraction layer in all of the cases grows linearly. The versions with 100-pieces are about 50.7% (Case 3) and 58.8% (Case 4) slower than the non-divided cases.

SAGA allows us to use different kinds of Data Grids. The speed degradation can be mitigated with SAGA by using faster storage resources in a mixture. The local file system is optimal for speed. The 100 piece version of Case 6 is about 34.5% slower than the non-divided version. The latency is decreased by using the SAGA and RNS solution.

Incidentally, in Figure 6.7, we encountered the fact that the results of iRODS are sometimes faster than that of local file. Strange as it may seem, such phenomena can certainly happen in the recent technology trends. Actually, Jeffrey Dean addressed “Numbers Everyone Should Know” saying network becomes faster than disk [96]. Therefore, we tried to evaluate performances of both network and storage with a “nuttcp” [97] application and a “dd” command in Linux, respectively. To obtain average values, each application ran ten times.

For the network evaluation, “nuttcp” tests resulted 856.6 Mbps between iRODS

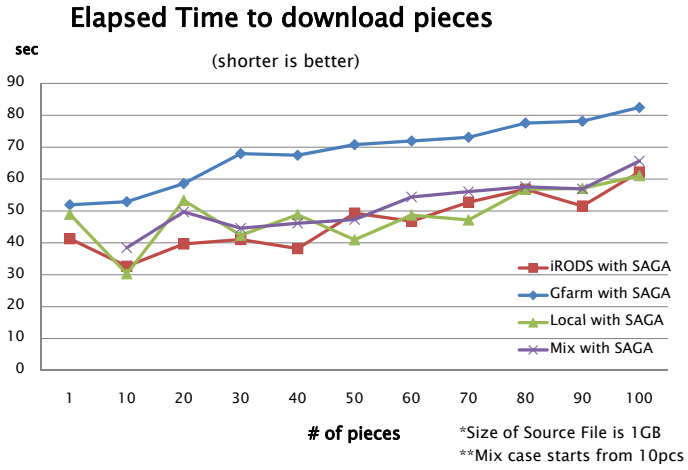


Figure 6.7: Performance evaluation results with SAGA

```
#!/bin/sh

for i in ‘seq 1 10‘
do
        echo $1
        nuttcp −n1G sg02
done
```

Figure 6.8: Script to execute a nuttcp test for network evaluation

server and client. Figure 6.8 shows an actual script to execute “nuttcp”. We found the virtual Ethernet device on the same physical machine acted ideally because the network is internally connected. For the storage evaluation, “dd” commands resulted 77.1 MB/s when reading a local file and writing the file to local file system. Figure 6.9 shows an actual script to execute “dd”. Both results showed that the network transfer rate was faster than the local file one in our environments.

Both virtual machines of iRODS server and client ran on a same physical machine. The performance of internal network achieved maximum transfer rate that was faster than local file-system transfer rate. When the iRODS server also has its own local storage the performance of iRODS became almost same as that of local file system. It is considered that the performance advantage of network over storage covered the overhead of iRODS service.

6.1.7 Discussion of UGI

Using UGI in this example is useful. Compared with Figure 6.2, part of the UGI application code is shown in Figure 6.10. The complexities of SAGA C++ and UGI Python are similar in this case. However, UGI allows us to implement the

```
#!/bin/sh

dd if=/dev/urandom of=/tmp/a bs=1M count=1000

for i in ‘seq 1 10‘
do
        echo $1
        dd if=/tmp/a of=/tmp/b bs=1M count=1000
        rm /tmp/b
done
```

Figure 6.9: Script to execute a dd command for storage evaluation

```
  // Open RNS directory
u = argv[1]

// get RNS junctions
rns_jncs = ugi.file.list(u)

// get physical locations and md5 checksum values
img_urls = []
img_md5s = []
for rns_jnc in rns_jncs:
   f_url = u + ”/” + rns_jnc
   epr = ugi.file.get_epr(f_url)
   md5 = ugi.file.get_metadata(f_url,”−k=checksum”)
   img_urls.append(epr)
   img_md5s.append(md5)
```

Figure 6.10: A part of UGI application example

file recovery functions easily. One of SAGA's limitations is that SAGA cannot transfer files among different kind of Data Grids directly, as mentioned in Section 3.3. When we encounter a checksum error during a program's execution, it is easy to recover the file by directly copying the piece from the source data to the desired target storage. For example, if we assume the source data is `DataGrid-s://path-A` and the target is `DataGrid-t://path-B`, we can write one function within the checksum error handling:
ugi.file.transfer_copy(”`DataGrid-s://path-A`”, ”`DataGrid-t://path-B`”)

In terms of speed, using UGI or Python is slower than using SAGA C++. Using the results of Section 5.1.2, the UGI application is 14.2% slower than SAGA C++. This indicates that UGI takes roughly an additional 3 to 10 seconds according to the results of Figure 6.7. That is a reasonable trade-off between performance and the useful functions.

6.2 Particle Therapy Simulation (PTSim)

We created a UGI-based Web application for particle therapy simulation. As already mentioned, UGI is implemented based on SAGA and provides supplemental and extended functions that are not part of SAGA. The prototype of the Web interface allows users to request most of the operations needed for their work. This Web interface shows the UGI possibility that non-Grid applications working on local resources are portably exported to distributed resources over the Grid resources.

6.2.1 PTSim Background

Geant4 is a toolkit for the simulation of the passage of particles. Geant4 is also applicable to other general simulations of radioactive-particle-tracking processes. The application of the Geant4 simulations to particle therapy using proton and ion beams is one of the medical activities in high energy physics. The PTSim system includes a graphical interface supporting collaboration among several medical centers studying particle therapy. The detailed simulations for human bodies require extensive computations and significant CPU resources. Parallel processing is useful to produce results with improved statistics and analyses. Here are some of the use cases in particle therapy simulations related to Grid computing:

Distributed job execution

To calculate more accurate statistics, it is often effective to divide a simulation into many smaller tasks if multiple clients can be applied to the calculations.

Secure job execution

Security has two relevant aspects. First, people who are working in hospitals typically have restricted to access the Internet. Second, patients' personal information must be kept secure from unauthorized personnel.

Spatial requirements and storage capacities

Large-scale data for such applications as CT scans using DICOM (Digital Imaging and Communication in Medicine) [98] which is a standard format for the patient data scanned by medical imaging systems, need to be stored in the hospital for treatment planning purposes.

6.2.2 PTSim Web Interface

The PTSim application requires tens of millions of particle ray tracings for a human body model. This requires thousands of jobs to be processed simultaneously to obtain the results in reasonable elapsed times. The PTSim Web interface is developed using the Django framework, which is convenient for Web development

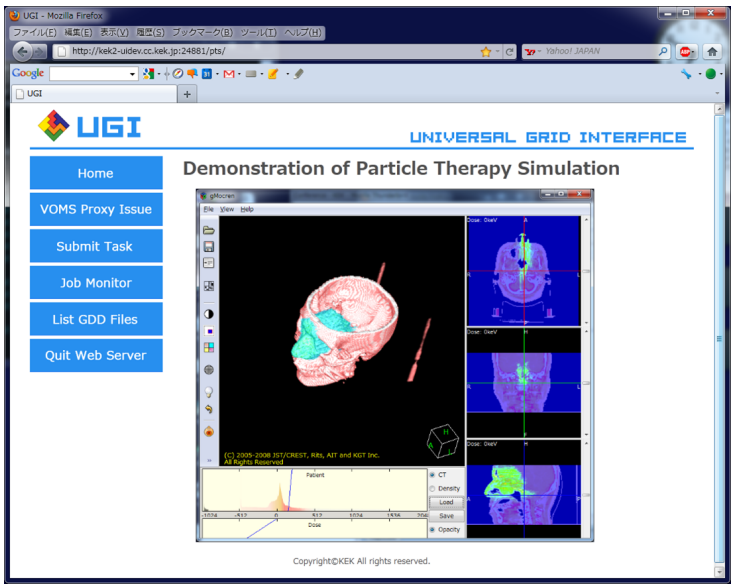


Figure 6.11: Application for PTSim work bench

in the Python language. The built-in Django Web server runs on the local host. An end user desktop PC browses it as shown in Figure 6.11.

The available kinds of Grid middleware supported in the current testbed are NAREGI, Globus, gLite for computing resources and GridFTP servers, Gfarm storage, and iRODS storage. After logging in and registering a proxy for NAREGI, multiple jobs described within a specific task script can be submitted to these kinds of middleware and the job status is updated in the database. Each job transfers its output file to a GridFTP server. The output files are processed on the user desktop PC for the graphic display, and all of these actions can be performed over the Internet.

Chapter 7

Applied Study

This chapter shows the applied study to utilize distributed resources. Swarm intelligence is one of approaches to provide a fault tolerant and efficient means of transferring data in a dynamic environment.

Swarm intelligence is inspired primarily by observations of the collective behavior of social insects in addressing complex distributed problems. The basic idea is that each member of the swarm has simple rules that govern its behavior, but the interaction among the members of the swarm can be used to tackle problems that are difficult to solve with complicated numeric methods. In this chapter we investigate the problem of data distribution between a client and server in a dynamic environment. We regard each download from the server to the client as a single member in a swarm. The member's behavior is simply to reliably download a data file. Each member can communicate with other members to allow the swarm to settle on the best set of servers to download the data from based on the current status of the environment.

Some research work generates large numbers of small files and then interacts with the generated small files in groups. For example, the T2K ND280 [99, 100] group consists of hundreds of researchers in 12 countries and 62 research institutes, all of whom must reliably download their own data files. They are using iRODS, which provides each user with a virtual file system that maps to distributed storage systems. iRODS has been developed by the Data Intensive Cyber-Environments (DICE) [101] team and collaborators and is based on more than a decade of experience with distributed data management systems ([20]). Different iRODS installations can be federated together to provide a larger virtual file system while allowing each member of the federation complete control over access and management of their own iRODS. This approach also allows client applications to interact with the data. Our implementation uses iRODS i-commands to download files from each system.

7.1 Ant Colony Optimization

Ant Colony Optimization (ACO) algorithms [102] are a type of swarm intelligence. They are based on the behavior of foraging ants in which individual ants search in a seemingly random manner for food. As an ant searches it leaves pheromone or scent that records on its discovery of a food source and the path used during its return to its nest. The amount of the pheromone reveals information about the nature of the food source. Subsequent ants follow the pheromone trail and also reinforce it when they return to the nest with food. There may be multiple trails to the food source, but after some time the ants will converge on the most direct path between the source and the nest. This is due to the evaporation of the pheromone, since longer paths will have weaker intensities of pheromones and will be less likely to be followed.

In solving complex problems ACO algorithms use computational agents (representing the ants) that perform simple tasks. Each agent constructs a candidate solution that is communicated to other agents via a probability (the pheromone element) that is based on the components used to construct the solution. For example, in the travelling salesman problem the probability is based on the edges between cities. Each probability contains a weight based on the heuristic information for the current problem. The weight represents the evaporation factor and reduces the probability for each ant's solution by a defined amount. The role of the weight is to eliminate local or intermediate solutions and reinforce the global or true solution.

7.2 The Data Distribution Problem

A common problem in almost any field that requires the processing of quantities of data is the movement of data from the storage systems to the computational systems where the data can be processed as quickly and reliably as possible. The problem is compounded by the dynamic nature of the environment in which the client is operating. The data collected by a central resource can rapidly become stale. The activity of each server can vary over time, the network activity can vary over time and the activity of each client can vary over time. In some cases network status information is coupled with server information through a broker service to guide the client to the best server [103]. However, these services require each server to publish the necessary information in order for the clients to make decisions. Since the servers cannot anticipate all of the needs of each client, it is possible that crucial information for a client will not be published by the server.

We argue that such priori information, although necessary, is actually encoded in the 'full' transfer rate from client to server. The 'full' transfer rate is simply the time taken for the client transfer application to complete a transfer. This includes the overhead of staging the data onto a disk on the server and finalizing the transfer

on the client (such as calculating a check-sum for the downloaded data). We believe that this is a better metric since the client is often interested not only in transferring the data as quickly as possible, but also in using the data as quickly as possible.

7.3 Related Work

7.3.1 ACO Related Work

The main area where swarms and ACO algorithms have been used is in optimizing distributed computational processing [104, 105, 106]. In such cases Particle Swarm Optimization (PSO), which is based on the flocking behavior of birds can optimize computational job submissions to the most efficient and least loaded nodes. Ant Clustering Algorithms (ACA) have been used to address the problem of clustering data in which related data should be clustered together (or co-located) for more efficient access ([107, 108]). Data clustering is crucial for data mining where the data is studied for patterns and relationships.

As in the case of ACO an ACA agent possesses simple behaviors and the interactions between agents allow complex problems to be solved. The ACA was based on studies of ant cemeteries where worker ants sort the deceased ants according to their size and function. Each ant works individually to arrange the dead ants in its local vicinity into a uniform group. Global sorting is done by the deceased ants on the edges being sorted by the neighboring worker ant. The ACA works by having each agent sort the data within a restricted vicinity (typically a 3×3 grid) so that all the data within that vicinity is of the same nature (where the nature is defined by the current problem). The data on the edges between neighboring agents is sorted first by one agent and then by the other (and then by their neighbors until they match a pile). The final result is clusters of data with similar properties.

In the area of data distribution using swarms the work by Peterson and Sirer [109] investigates the problem of data distribution in a peer-to-peer network. Peer-to-peer networks operate in a non-privileged manner where there is no central server and each client is also a server of data. The paper described the development of Antfarm, a system that manages the bandwidth usage of each server for the optimal download rates by a swarm of clients. The Antfarm system consists of coordinators that use information from seeders and peers to control the bandwidth for the peers downloading data such that the data is downloaded to members of the swarm in the most efficient manner possible. The system also encourages downloads between peers to distribute the bandwidth requirements. This study differs in that the main focus of this work is the problem of optimizing upload and download performance in a client-server environment.

Ant Colony Optimization has been studied in peer to peer networks by Wang Zhao and Hu [110] who looked at the problem of data replication optimization so

that the data would be replicated to the peers that could make the most efficient use of the available resources. Each agent used the host latency, storage space and bandwidth as ingredients in the pheromone to determine the best placement for all of the data on all of the available hosts. The ACO then globally optimized the placement of the replicas by allowing each agent to choose a placement based on the previous agent's attempt. The placement was governed by the strength of the pheromone at each site. The optimization finished when the agents did not return a better arrangement. The placement of replicas has similarities with the work described in this paper except that the global optimization was done only one time.

7.3.2 Compared with Other Services

There are some other services for redundancy mechanisms in distributed data systems. The Contents Delivery Network (CDN) [111] and load-balancing are well-known examples.

A typical CDN application tries to find hosts are located at the fewest number of hops from the client and it selects the best host to optimize the download performance. A typical application of load-balancing is to provide a single Internet service from multiple servers. However, both cases require installations of software and services on the server side to manage the client load. Our approach does not require the servers install any software. Our approach is client-based system and there is no impact or changes on the server-side. Also, our approach imposes no overhead on the server-side, but it offers advantages to the clients to get the data more efficiently when it resides in a number of different locations. As long as a user has access to the data (either through iRODS or any other file system) then the user can use the ACO to obtain the data in an optimal way (as long as there are multiple copies of the data).

7.4 Pheromone Definition

The essential component of the ACO is the pheromone. Ants collect their food using their pheromone. The environment around ants is similar to the environment of clients collecting data from servers (Figure 7.1).

The pheromone indicates to the agents which are the more promising paths to use in constructing a solution to a problem. In our case the pheromone is a metric of the viability of the server to serve the data to the client in as short a time as possible. To encourage a quick convergence to a solution we first determine the server availability to handle requests to download data. The availability is dependent on the load on the server and on the network.

A 'ping'-like application that sends a light-weight query to the server can assess the viability of the server (We describe an example of a 'ping'-like application

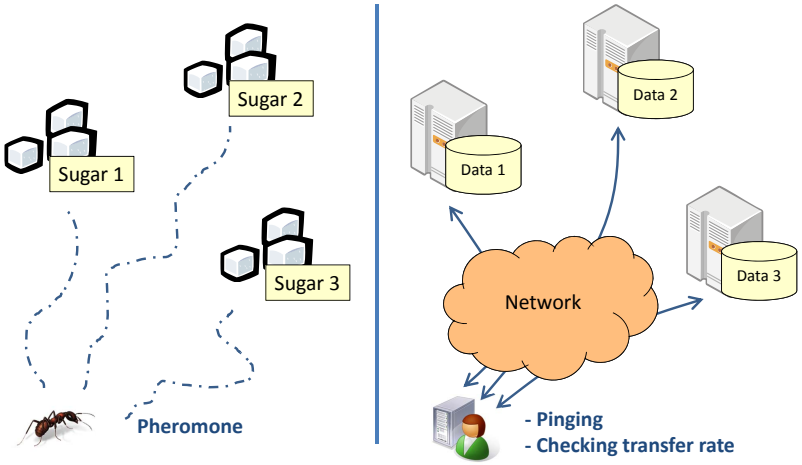


Figure 7.1: Environments of ants-foods and clients-data.

in Section 7.5). The servers would then be ranked according to their responses to the 'ping'. It is important to point out that the application should ping the server application that serves the data and not the server itself since the application may be overloaded or down whilst the server is only moderately loaded or up and still able to respond quickly to a ping request.

However, ranking servers according to their response times to a lightweight query is not sufficient to optimize the download performance. It is possible that a server may respond quickly to a lightweight query, but may be either unable to serve the data due to some component of the storage system being offline (in the case of a compound storage system with a disk cache and a tape store where it is possible the tape store may be offline), or the storage resource being very busy (possibly due to high fragmentation in the case of disk storage systems). To address such situations we devised a pheromone element based on a transfer rate metric. The rate is the inverse value of the complete download time measured from the time that the download application starts to the time that it finishes. This rate is necessarily smaller than the actual transfer rate because it includes the download time for the server to fetch the data from its system, serve it to the client, and any time required for the client download application to prepare the data for use. We do not care about the actual transfer rate in the network but we only consider the inverse value of the complete download time. Therefore, we redefine the inverse value of the complete download time as the 'TransferRate'.

7.4.1 Pheromone Element

A pheromone's current value is based on the historical pheromone values so the base pheromone element must be defined first. We define the set $\mathcal{S}$ that includes all of the servers we want to use. M is defined as the number of servers in $\mathcal{S}$, and

p_i expresses the pheromone element of $s_i \in S$, $0 \leq i \leq M$. The pheromone element is given by:

$$
\begin{aligned}
n &= 1: \\
& p_i(1) = \frac{(CurrentTransferRate)_i}{\sum_{i=0}^{M} (CurrentTransferRate)_i} && (7.1) \\
n &> 1: \\
& p_i(n) = \frac{(CurrentTransferRate)_i}{\sum_{i=0}^{M} (PreviousTransferRate)_i} && (7.2)
\end{aligned}
$$

where $CurrentTransferRate$ is the rate used by the download application to start and complete for a given server. The $PreviousTransferRate$ is the rate taken by the download application for previous transfers. The n corresponds to the number of files downloaded from a given server. The first time a file is downloaded no prior history exists and the pheromone element $p_i(1)$ appears as a weight of the current TransferRate as shown in the first equation (Eq. (7.1)).

The pheromone element value is calculated immediately after downloading a file and is stored in a information file. This will be explained in Algorithm 7.2 in the next section.

7.4.2 Pheromone

Now we can define the signature of the pheromone elements. We simply call it "pheromone". The h is given as the number of p_i histories. The capital P_i expresses the pheromone and it is given by:

$$
\begin{aligned}
1 \leq \quad n &\leq h: \\
& P_i(n) = \sum_{k=0}^{n} p_i(k) && (7.3) \\
n &> h: \\
& P_i(n) = \sum_{k=n-h}^{n} p_i(k) && (7.4)
\end{aligned}
$$

The pheromone value is calculated by reading the information file just before downloading a file. This is described in Algorithm 7.1 in the next section.

7.5 Algorithm

Our approach selects the best server using pheromone information before a client tries to download a file from a server. It also requires an information file to record each server's information and to update the information file immediately after the download. Our ACO agent uses these algorithms:

i	Server Name	iping Time	Download TransferRate	$p(6)$	$p(7)$	$\cdots$	$p(10)$
0	Host01.kek.jp	3.443	42.432	0.244	0.244	$\cdots$	0.244
1	Host02.kek.jp	5.165	28.288	0.172	0.172	$\cdots$	0.172
2	Host03.kek.jp	4.238	34.476	0.209	0.209	$\cdots$	0.209
3	Host04.kek.jp	4.768	30.645	0.148	0.148	$\cdots$	0.148
4	Host05.kek.jp	8.106	18.026	0.225	0.225	$\cdots$	0.225

Table 7.1: The example of an information file (e.g. n=10, h=4)

7.5.1 Algorithm to Select the Best Server

The best server is obtained by using Algorithm 7.1. An example of an information file is shown in Table 7.1. We created the command 'iping' as an example of a 'ping'-like application that checks the responses from the servers. In this example, the units for the iping values of Time and TransferRate are msec and MB/sec, respectively. The set $\mathcal{S}$ has all of the servers that are listed in the information file, $infoText$. The $infoText$ file also has the historical pheromone element values (p_i) for each server that were previously defined in the equations(Eq. (7.1), Eq. (7.2)). Reading P_i means to read the required p_i from $infoText$ and calculate P_i as defined in the equations(Eq. (7.3), Eq. (7.4)). The n corresponds to the number of downloads in progress at that time.

The iping Boundary Time ($ipBT$) is a fixed reference value for the iping results and is set at the hypothetically best response time. This helps to filter out servers in the $srvList$ that have unacceptable response times (either because they are busy and cannot respond within an acceptable time or because they are offline).

7.5.2 Algorithm to Update the Information File

While executing a download, the given server becomes the $bestServer$ that is selected by Algorithm 7.1. The $infoText$ file is then updated immediately after each download is completed. The $infoText$ file is updated using Algorithm 7.2. $TransferRate_{new}$ is the TransferRate of the current download from the $bestServer$.

Using only Algorithm 7.2 sometimes ignores possibilities that the non-recorded servers become predominant. As necessary, we use another Algorithm 7.3 to download data from all servers to check the performance of non-recorded servers. For example, when we want to check the performance of all servers once every five times, updating $infoText$ in the first four times uses Algorithm 7.2 and updating in the fifth time uses Algorithm 7.3 instead of Algorithm 7.2. However, Algorithm 7.3 causes network traffic. We need to consider the trade-off between accuracy and network traffic when using Algorithm 7.3.

Algorithm 7.1 Select the best server with pheromone

1: open file $infoText$
2: create the set $\mathcal{S}$
3: close file $infoText$
4: **for** each $serverName\ s_i$ in $\mathcal{S}$ **do**
5: execute iping to s_i
6: $tp_i \leftarrow$ response time of s_i iping
7: add tp_i to $ipingList$
8: **end for**
9: $tp_{min} \leftarrow \min(tp \in ipingList)$
10: **for** each $serverName\ s_i$ in $\mathcal{S}$ **do**
11: **if** $tp_i \leq (tp_{min} + ipBT)$ **then**
12: add s_i to $srvList$
13: **end if**
14: **end for**
15: open file $infoText$
16: **if** $n > 1$ **then**
17: **for** each $selectedServer\ ss_i$ in $srvList$ **do**
18: seek the location of ss_i information
19: read P_i from $infoText$
20: add P_i to $PList$
21: **end for**
22: **else**
23: **for** each $selectedServer\ ss_i$ in $srvList$ **do**
24: seek the location of ss_i information
25: read $TransferRate_i$
26: add $TransferRate_i$ to $trList$
27: **end for**
28: **for** each $selectedServer\ ss_i$ in $srvList$ **do**
29: calculate $p_i(1)$ with $TransferRate_i$ and $trList$
30: add $p_i(1)$ to $PList$ as $P_i(1)$
31: **end for**
32: **end if**
33: close file $infoText$
34: $P_{max} \leftarrow \max(P \in PList)$
35: **for** each $selectedServer\ ss_i$ in $srvList$ **do**
36: **if** P_i is equal to P_{max} **then**
37: $bestServer \leftarrow ss_i$
38: break;
39: **end if**
40: **end for**
41: return $bestServer$

7.5.3 Comparison with Traditional Method

One of the traditional methods is just using the best transfer rate from the previous session. The algorithm using this method can be implemented by using the best

Algorithm 7.2 Update the information file

execute download from $bestServer$
$TransferRate_{new} \leftarrow TransferRate$
open file $infoText$
create the set $\mathcal{S}$
for each $serverName\ s_i$ in $\mathcal{S}$ **do**
    seek the location of s_i information
    read $TransferRate_i$
    add $TransferRate_i$ to $trList$
end for
close file $infoText$
calculate p_{new} with $TransferRate_{new}$ and $trList$
open file $infoText$
seek the location of $bestServer$ information
update $TransferRate_{bestServer} \leftarrow TransferRate_{new}$
add $p_{bestServer} \leftarrow p_{new}$ to $infoText$
remove p_{oldest} from $infoText$
close file $infoText$

Algorithm 7.3 Update the information file with downloading data from all servers

open file $infoText$
create the set $\mathcal{S}$
close file $infoText$
for each $serverName\ s_i$ in $\mathcal{S}$ **do**
    execute download from s_i
    $TransferRate_i \leftarrow TransferRate$
    add $TransferRate_i$ to $trList$
end for
open file $infoText$
for each $serverName\ s_i$ in $\mathcal{S}$ **do**
    calculate p_i with $TransferRate_i$ and $trList$
    seek the location of s_i information
    update $TransferRate_i$
    add p_i to $infoText$
    remove $p_{ioldest}$ from $infoText$
end for
close file $infoText$

transfer rate with the same algorithms (Algorithm 7.1, 7.2, 7.3) instead of using P_i and p_i. This method seems to be simple, but there is no difference in the algorithms. In addition, with this method we cannot correct the historical information, as when using h in our approach. The results of the traditional method are shown in Section 7.7.

7.6 Simulation

We created a simulator to study the behavior of ACO-based data transfers. The simulator provided a controlled environment within which it was possible to study different types of scenarios.

7.6.1 Model

For the simulation we modeled two typical scenarios for downloading data in a distributed environment. The model assumed the data set spanned five different servers.

- *Phased Degradation*. In this case the performance of each server degrades over time as shown in Figure 7.3. After the first file has been transferred the first server's performance degrades. The other servers' performance also degrades as they complete transfers. After seven transfers the performance improves for all of the servers, so they return to their optimal performance status after 12 files have been transferred. This situation is fairly common in distributed environments when clients start working in lock-step among the servers. Such a situation may appear when a group of clients start to use one system until its performance becomes unacceptable and they search for a new server for their downloads.

- *Random Degradation*. In this case the performance of each server degrades randomly over time as shown in Figure 7.5. The performance of the first server degrades after 10 transfers and then improves and degrades again after 17 transfers. The second server degrades after seven transfers, returns to optimal performance after 13 transfers and then degrades after 20 transfers. The other servers follow similar patterns. This situation models a more random access pattern where there is no coupling among the performance of the servers.

7.6.2 Procedure

The simulation first required the preparation of input data for the ACO-based data transfer application. The models were used to generate several information files (Figure 7.2). The server conditions for the two scenarios were defined in advance in each information file. The simulator runs by reading each information file. These information files contain rows of numbers according to the following schema (the example information file is already shown in the Table 7.1):

- server name
- 'ping' time
- download transfer rate

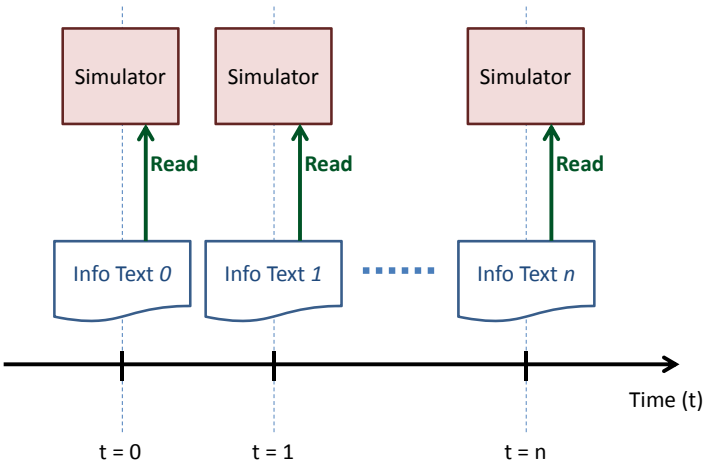


Figure 7.2: Simulator uses several information files.

- upload transfer rate
- $p_i(k)$

Each row corresponded to the download information for a given file for a given server. For these simulations the download information was based on the iRODS *iget* download application. The simulation program read in a data set and ranked the servers according to the 'ping'-like information. The 'ping' information was based on the results of the *iping* command for iRODS. This determined the initial selections for which agents would use the hosts.

Each agent in the simulation program then used the best host on the list and started to read the simulation data (which included simulated download rates for the servers). The pheromone was then computed with Eq. (7.1) using the information from all of the available hosts that had completed their first download. Each agent ready to perform a download selected the available host with the best pheromone and updated the pheromone value after the download using Eq. (7.2). This procedure continued until the simulated data was exhausted.

7.7 Simulation Results

The results of the simulation are shown in Figure 7.3 for the *Phased Degradation* model and in Figure 7.5 for the *Random Degradation* model. Both models are using four pheromone histories ($h = 4$).

7.7.1 Phased Degradation

Figure 7.3 shows the first simulation results corresponding to the *Phased Degradation* model. The upper figure shows the transfer rate for each server without the

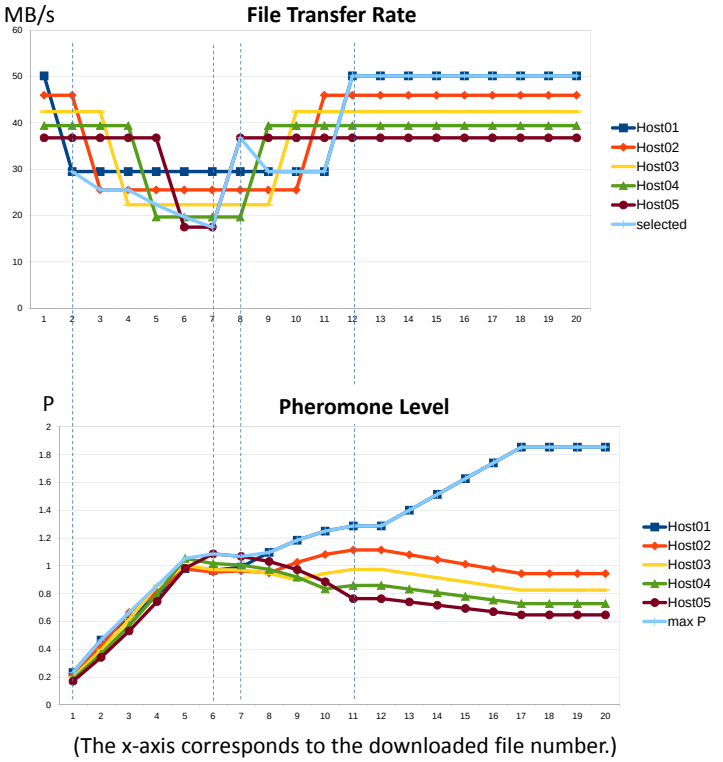


Figure 7.3: Transfer-Rate and Pheromone for the phased degradation.

ACO-based download. The *selected* curve corresponds to the ACO-based download. The lower figure shows the pheromone value for each host with the *max P* curve corresponding to the ACO-based download case. In this case of rapid degradation of the servers the ACO-based approach performed well. The pheromone is based on history information and there is always a delay between the response of the ACO-based download and the performance of the server.

In the initial stages the best server rapidly becomes the worst server resulting in the ACO-based approach tracking the degradation of the servers. However, as the servers improve in performance the ACO-based approach gradually improves. Clearly, this situation is a troublesome case, but realistic, situation and it is encouraging that the trough in the performance is steeper than that for each server, indicating that the algorithm is doing well in a bad situation. The lower graph shows consistently that the value of *max P* corresponding to the best path consists of those hosts with the best pheromone value at that time.

We also simulated this situation with the same data in the traditional method described in Section 7.5. Figure 7.4 shows the results from using slightly lower-performance hosts compared with our approach.

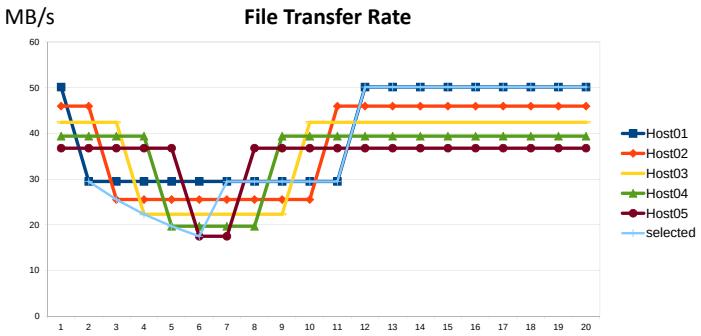


Figure 7.4: Transfer-Rate in the traditional way.

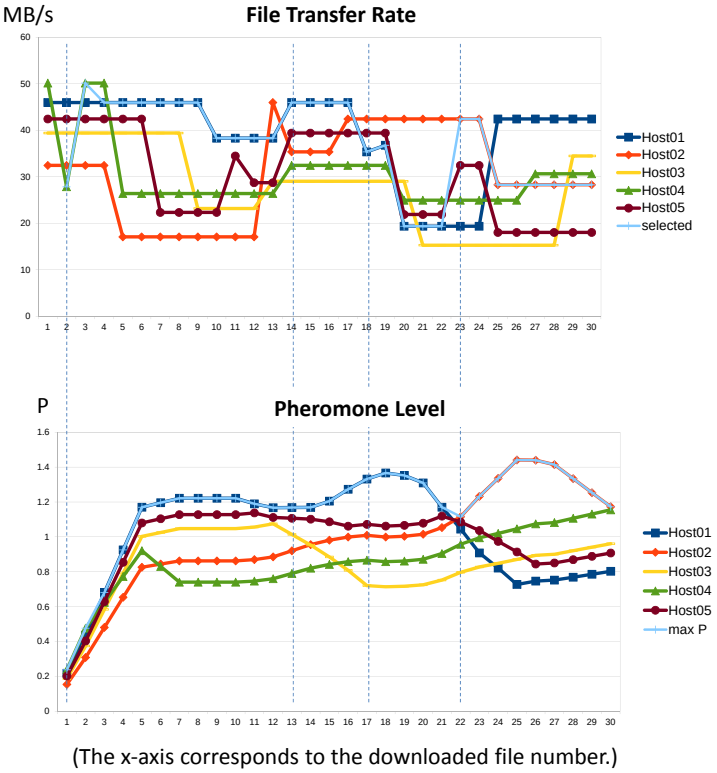


Figure 7.5: Transfer-Rate and Pheromone in the random degradation model.

7.7.2 Random Degradation

Figure 7.5 shows the simulation results for the *random degradation* model. This case clearly shows that the results from the ACO-based approach shown in the *selected* curve out-perform those based on any individual server. The visible dips at the beginning and end of the transfer period are an artifact of the pheromone having to rely on historical information. The pheromone for the best hosts shown in the *max P* curve in the lower figure consistently corresponds to the best host at that time.

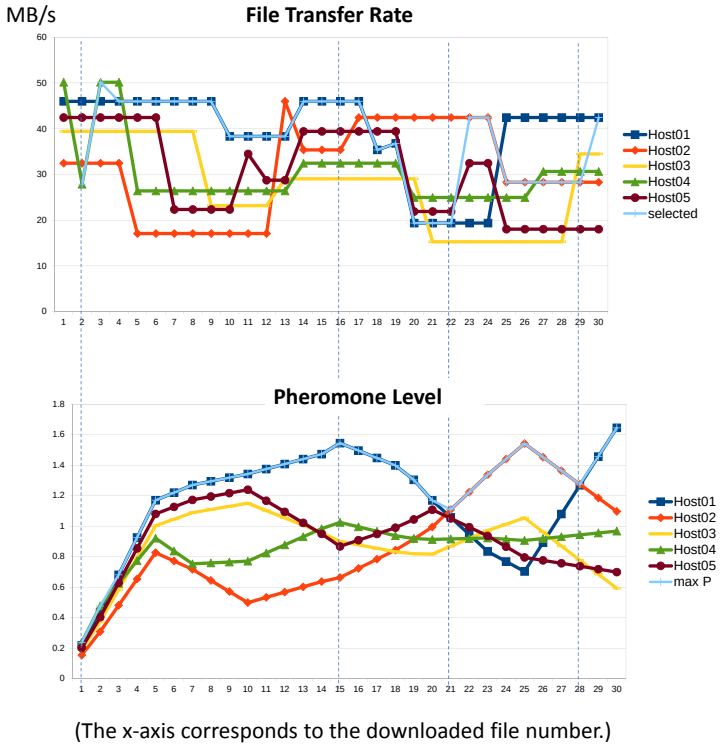


Figure 7.6: Random degradation model with Algorithm 7.3.

Moreover, checking all server performances with Algorithm 7.3 can improve the accuracy to select servers. We checked the performance of all servers once every five times in the above random degradation model and the results became a little better as shown in Figure 7.6. The pheromone level in the beginning of the results was not so different from Figure 7.5 but this case was finally optimized to select the best server in the end.

7.8 Test Implementation

We took the ACO-based application and used it for a real test-setup consisting of three distributed servers: one in the UK, one in the USA and one in Japan. We used the *iget* application of iRODS to download files from each system. The implementation required the development of scripts to provide the functions needed to implement the ACO-based downloads. These consisted of:

- iping (new i-command for iRODS). This was needed to perform the light-weight queries of the servers r to determine the server rankings.
- iping.py (script to drive the iping command). This script wrapped the iRODS icommand with iget.py.
- iget.py (script using the original iget command). This script implemented the ACO agent (as the ACO-based download algorithms) and called the

iRODS *iget* command.

7.8.1 iping/iping.py

Currently iRODS does not have a command like *ping* that can be used to check the server availability. We created the *iping* application that calls the iRODS server and gets the echo outputs from the server.

The *iping.py* can specify the iRODS host with the option "-H" and iping the server. That is because the *iping* command can execute only on the server that is specified in the client's iRODS configuration file (.irodsEnv). All i-commands should follow the information in the .irodsEnv file so we avoid including the option specifying a server in the *iping* command, instead, the *iping.py* script takes charge of the options. The *iping* application also includes *"ping_to_all()"* function that can execute *iping* to all servers specified in a configuration file. This function is useful for checking all server availability just before executing *iget* commands. After executing the *iping* command invoked by the *ping_to_all()* function, the *iping.py* updates the ranking of the servers.

7.8.2 iget.py

The scripts for downloading (*iget.py*) a file execute the following steps:

1) execute *iping* for all of the servers
2) read the configuration file
3) select the best server
4) execute *iget* for a file
5) get the current *iget* transfer rate
6) calculate $p_i(k)$
7) update the transfer rate and $p_i(k)$

The steps except for 4) executing *iget* are our ACO agent tasks. The best server is selected in exactly the same manner as in the simulation. First, the servers are ranked according to their ping responses, and then the server with the best P is chosen.

7.9 Test Results

The test-setup was highly distributed and consisted of three iRODS servers: one located at Queen Mary University of London (QMUL), UK, one at Louisiana State University (LSU), USA and one at KEK in Japan.

The tests were performed at KEK which is regarded as the local host and so the performance would be much better within unloaded servers. To address this we artificially adjusted the ranking results from the *iping* to give KEK the lowest ranking. The results are shown in Figure 7.7. In this example, the same pheromone

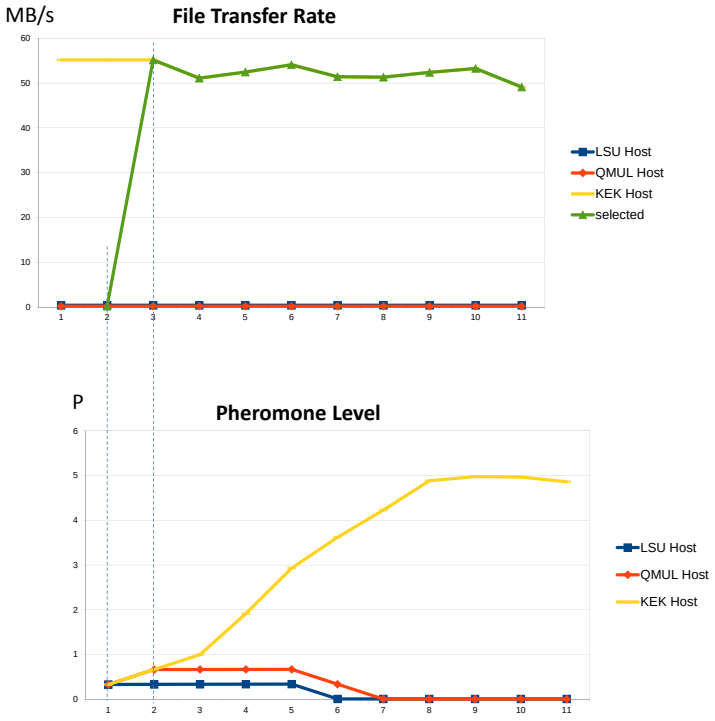


Figure 7.7: Transfer-Rate and Pheromone in the actual case.

values are given as for the initial pheromone and the pheromone history(h) is 4. The first file was downloaded from LSU and the next from QMUL. In both cases the pheromone value was low (with the initial value set to one third of the corresponding pheromone for each server). The third file was downloaded from KEK resulting in a much higher pheromone value. Subsequent agents quickly settled on this host for downloading the data reinforcing the pheromone value for the KEK server. This demonstrated that the ACO-based approach was able to quickly find the optimum performance in a real environment.

We ran this demonstration with one client and three servers. This approach can scale easily since each client independently checks the servers. Therefore, we can use this approach in real environments (as mentioned in the introduction).

7.10 Discussion about UGI Use

The algorithms presented in this chapter can be used in our UGI implementation. In particular, Python implementations can be easily migrated to the UGI implementation. In addition, using UGI provides the advantage of storing the information. We can register any kind of information (or metadata) into the RNS catalog service, so we are freed of worrying about a local text file to contain the information. As we discussed in Chapter 3 and 4, UGI can access the storage resources on different kinds of the file-system middleware. We can store the source data in distributed storage resources in different kinds of Grid middleware and obtain the data by using our ACO approach (Figure7.8). Our ACO approach and UGI do not require any changes for the server side. Therefore, our ACO approach and UGI are a good combination to utilize resources in different kinds of Grid middleware.

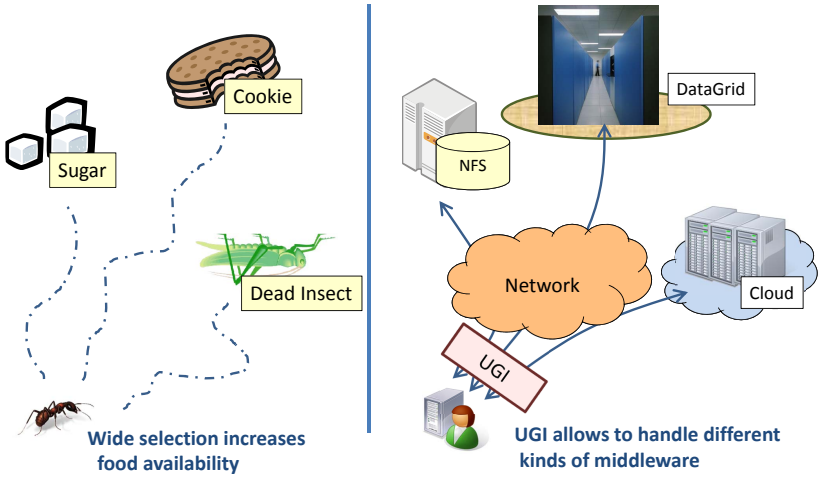


Figure 7.8: ACO approach in different Data Grids with UGI.

Chapter 8

Conclusion

It is essential that we effectively share computing and storage resources to enhance our research. Grid computing and Cloud technologies are the main examples of distributed resources using the Internet. Recent scientific challenges require the worldwide collaboration of researchers sharing their resources, such as computational system, large amounts of distributed data, software, and knowledge. However, we frequently encounter difficulties in exchanging or sharing resources based on different kinds of middleware when the load of the computing and the storage use are unbalanced. We studied the software-abstraction layer and developed the UGI architecture for multiple kinds of Grid and Cloud middleware to help end users and application engineers. UGI is implemented based on SAGA and provides supplemental and extended functions.

Our tests have shown that job submissions can be executed in the UGI-based user environment with different Grid resources. UGI allows us to address our requirement that the applications should not need to be changed for new or unknown middleware. We have also created and tested a simple way to execute the jobs based on the HENP libraries. For file manipulation, we have demonstrated that applications can access different kinds of file-system middleware and Data Grids. The application allows us to handle them as a completed file, even if a large file is divided into pieces and the divided data is stored on different Data Grids. We verified to access iRODS, Gfarm and local file system via UGI. With our approach, data sharing is more expandable and flexible. We confirmed that there is no need to change the application itself to access files on the multi-file-system middleware. We tested managing files distributed among heterogeneous Data Grids by using the RNS application, proving that a UGI-based application can retrieve the location information of the files distributed among different kinds of Data Grids, and that it can access the distributed files as well as the local file systems without worrying about the underlying Data Grids. Our tests showed that not only our application can access files in the multiple Data Grids, but also that the physical file locations and other metadata associated with each file can be shared with RNS.

For use with applied tools and applications, we demonstrated reliably manag-

ing files, PTSim, and ACO. The method for reliably managing large files worked well with different kinds of Data Grids using SAGA and RNS. We showed how to split a large file and store its MD5 checksum value as metadata in the RNS catalog service. We also showed how the application can test all of the checksum values and then combine the pieces that were distributed among the different Data Grids. Our tests showed that the physical file locations and MD5 checksum values associated with each file can be shared by using RNS. Our tests also showed that the speed degradation can be mitigated by using faster storage resources in a mixture. The second applied tool is a UGI-based Web application for PTSim. The prototype Web interface allows users to request most of their PTSim job operations. UGI also make it possible for non-Grid applications to use local resources that are portably exported to distributed resources over the Grid. For an approach inspired by swarm intelligence, we created a simulator using our ACO-based approach and obtained results that proved our approach works well. This approach can provide a fault tolerant and efficient means of transferring data in a dynamic environment. We implemented this approach using several iRODS servers as the distributed file systems. This approach is easy to apply to different kinds of Data Grids with UGI.

We can effectively utilize various computing and storage resources with our implementations and solutions. The challenges of today's researchers who need to collaborate with geographically distributed colleagues and computing and storage resources can be overcome. We believe that our studies of the resource federation can greatly boost their usability for e-Science.

Bibliography

[1] "Welcome to GGUS - the Helpdesk," Online, https://ggus.eu/pages/home.php.

[2] B. G. and W. J., "Swarm Intelligence in Cellular Robotic Systems," in *Proc. the NATO Advanced Workshop on Robots and Biological Systems*, Tuscany, Italy, Jun. 1989.

[3] "NeSC: National e-Science Centre," Online, http://www.nesc.ac.uk/.

[4] "Defining e-Science (NeSC)," Online, http://www.nesc.ac.uk/nesc/define.html.

[5] "Current Awareness Portal E742 - ARL (Japanese)," Online, http://www.current.ndl.go.jp/e742.

[6] I. Foster, "What is the Grid? A Three Point Checklist," Old Dominion University Digital Library Group, Tech. Rep., 2002, http://dlib.cs.odu.edu/WhatIsTheGrid.pdf.

[7] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[8] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200 – 222, Aug. 2001.

[9] "Globus," Online, http://www.globus.org/.

[10] "EGEE: The Enabling Grids for E-sciencE," Online, http://public.eu-egee.org/.

[11] "gLite: Lightweight Middleware for Grid Computing," Online, http://glite.web.cern.ch/glite/.

[12] "CERN: The European Organization for Nuclear Research," Online, http://public.web.cern.ch/public.

[13] S. Matsuoka, S. Shimojo, M. Aoyagi, S. Sekiguchi, H. Usami, and K. Miura, "Japanese Computational Grid Research Project: NAREGI," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 522–533, March 2005.

[14] “NAREGI Middleware download site,” Online, http://middleware.naregi.org/.

[15] “NII: National Institute of Informatics,” Online, http://www.nii.ac.jp/en/.

[16] “Globus Toolkit,” Online, http://www.globus.org/toolkit/.

[17] B. Jones, “EGEE - a worldwide Grid infrastructure,” August 2005, the 19th International Congress of the European Federation for Medical Informatics (MIE), Geneva. http://egee-intranet.web.cern.ch/egee-intranet/NA1/presentations/ppt-fbm/2005/MIE-2005.ppt.

[18] “KEK: High Energy Accelerator Research Organization,” Online, http://www.kek.jp/intra-e/.

[19] “IBM Platform LSF Product Family,” Online, http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/.

[20] “iRODS – the Integrated Rule-Oriented Data System,” Online, http://www.irods.org.

[21] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder, “A Prototype Rule-based Distributed Data Management System,” in *Proc. HPDC workshop on ”Next Generation Distributed Data Management”*, Paris, France, May 2006.

[22] “Gfarm – Grid Data Farm,” Online, http://datafarm.apgrid.org/index.en.html.

[23] “OGF – Open Grid Forum,” Online, http://www.ogf.org/.

[24] “SAGA: A Simple API for Grid Applications,” Online, http://saga.cct.lsu.edu/.

[25] K. Aida, “Grid in Cyber Science Infrastructure,” April 2009, iSGC 2009, Academia Sinica, Taipei, Taiwan.

[26] “RENKEI – REsources liNKage for E-scIence,” Online, http://www.e-sciren.org/index-e.html.

[27] “TeraGrid Archives,” Online, https://www.xsede.org/tg-archives.

[28] “Extreme Science and Engineering Discovery Environment (XSEDE),” Online, https://www.xsede.org/.

[29] M. Pereira, O. Tatebe *et al.*, “Resource namespace service specification (GFD-R-P.101),” GFS-WG, Tech. Rep., 2007, http://www.ggf.org/documents/GFD.101.pdf.

[30] H. Matsuda, “File Catalog Development in Japan e-Science Project,” GFS-WG, Tech. Rep., 2008, http://www.ogf.org/OGF24/materials/1403/OGF24-GFS-matsuda.pdf.

[31] T. Aso, A. Kimura, S. Kameoka, K. Murakami, T. Sasaki, and T. Yamashita, “GEANT4 Based Simulation Framework for Particle Therapy System,” in *Proc. IEEE Nuclear Science Symposium Conference Record*, Hawaii, US, Nov. 2007, pp. 2564–2567.

[32] T. Sasaki and S. Tanaka, “Comprehensive Software Suite for Particle Beam Simulation — Special Feature — Development of a Simulation Framework for Radiotherapy (Japanese),” *Japan Society for Simulation Technology*, vol. 28, no. 1, pp. 2–3, Mar. 2009.

[33] “Remote execution of applications,” Online, http://www.faqs.org/docs/linux_intro/sect_10_03.html.

[34] H. Gjermundrod, M. D. Dikaiakos, M. Stumpert, P. Wolniewicz, and H. Kornmayer, “g-Eclipse – an integrated framework to access and maintain Grid resources,” in *Proc. the 9th IEEE/ACM International Conference on Grid Computing*, TsukubaCJapan, Sep. 2008, pp. 57 – 64.

[35] D. Johnson, K. Meacham, and H. Kornmayer, “A middleware independent Grid workflow builder for scientific applications,” in *Proc. the 5th IEEE International Conference on E-Science*, Oxford, UK, Dec. 2009, pp. 86 – 91.

[36] R. Brobst, W. Chan *et al.*, “DRMAA - v1.0 Specification (GFD-R.022),” DRMAA-WG, Tech. Rep., 2004, http://www.ggf.org/documents/GFD.22.pdf.

[37] Goodale, Tom *et al.*, “SAGA - v1.0 Specification (GFD-R-P.90),” SAGA-CORE-WG, Tech. Rep., 2008, http://www.ggf.org/documents/GFD.90.pdf.

[38] “DRMAA-WG: Distributed Resource Management Application API Working Group,” Online, http://forge.ogf.org/sf/projects/drmaa-wg/.

[39] O. Tatebe, “Discussion of File Catalog Standardization,” GFS-WG, Tech. Rep., 2008, http://www.ogf.org/OGF24/materials/1403/intro.pdf.

[40] “Web Services Addressing 1.0 – Core,” Online, http://www.w3.org/TR/ws-addr-core/.

[41] N. Masahiro and T. Osamu, “Implementation of Resource Namespace Service [in Japanese],” *Information Processing Society of Japan (IPSJ)*, vol. 5, pp. 145 – 146, Mar. 2008.

[42] "Simple Object Access Protocol (SOAP) 1.1," Online, http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[43] T. Ishibashi, Y. Kido, T. Fukumoto, S. Seno, Y. Takenaka, and H. Matsuda, "A metadata management system for composing bioinformatics workflows," in *Proc. the 9th International Conference on Bioinformatics (InCoB)*, TokyoCJapan, Sep. 2010.

[44] M. Nakamura and O. Tatebe, "Load balancing of Resource Namespace Management Service (Japanese)," in *Proc. the Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP)*, SagaCJapan, Aug. 2008.

[45] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," in *Proc. the 1998 conference of the Centre for Advanced Studies on Collaborative research (CASCON 1998)*, Toronto, Canada, Nov. 1998, p. 5.

[46] M. Hedges, A. Hasan, and T. Blanke, "Curation and Preservation of Research Data in an iRODS Data Grid," in *Proc. The Third IEEE International Conference on e-Science and Grid Computing*, Bangalore, India, Dec. 2007, pp. 457 – 464.

[47] V. Muppavarapu and S. M. Chung, "Semantic-based Access Control for Grid Data Resources in Open Grid Services Architecture - Data Access and Integration (OGSA-DAI)," in *Proc. Tools with Artificial Intelligence, 2008. ICTAI '08. 20th IEEE International Conference on*, Dayton, OH, Nov. 2008, pp. 315 – 322.

[48] ARCS and JCU, "Hermes," Online, http://projects.arcs.org.au/trac/commons-vfs-grid/.

[49] A. Grimshaw, M. Morgan, D. Merrill, A. S. Hiro Kishimoto, D. Snelling, C. Smith, and D. Berry, "An Open Grid Services Architecture Primer," *Proceedings of the IEEE*, vol. 42, no. 2, pp. 27 – 34, Feb. 2009.

[50] J. Green, "An Implementation of the Resource Namespace Service Specification for OGSA-DAI," Master's thesis, The University of Edinburgh, 2008.

[51] "AMGA, ARDA Metadata Grid Application," Online, http://amga.web.cern.ch/amga.

[52] N. Santosa and B. Koblitza, "Metadata Services on the Grid," in *Proc. Advanced Computing and Analysis Techniques (ACAT)*, Berlin, Germany, May 2005.

[53] ——, "Distributed Metadata with the AMGA Metadata Catalog," in *Proc. the Workshop on Next-Generation Distributed Data Management HPDC-15*, Paris, France, Jun. 2006.

[54] "XQuery: A Query Language for XML," Online, http://www.w3.org/TR/2001/WD-xquery-20010215/.

[55] Z. Dadan, C. Zhebing, W. Jianpu, Z. Minqi, and Z. Aoying, "Different File Systems Data Access Support on MapReduce," in *Proc. Computational Intelligence and Software Engineering (CiSE)*, Wuhan, China, Dec. 2009, pp. 1 – 4.

[56] "Hadoop," Online, http://hadoop.apache.org/.

[57] "Kosmos File System," Online, http://kosmosfs.sourceforge.net/.

[58] H.-R. Mizani, L. Zheng, V. Vlassov, and K. Popov, "Design and Implementation of a Virtual Organization File System for Dynamic VOs," in *Proc. The 11th IEEE International Conference on Computational Science and Engineering (CSE)*, Sao Paulo, Brazil, Jul. 2008, pp. 77 – 82.

[59] H. E. Wedde and J.-O. P. Siepmann, "A Universal Framework for Managing Metadata in the Distributed Dragon Slayer System," vol. 2, pp. 96 – 101, Sep. 2000.

[60] D. Feng, J. Wang, F. Wang, and P. Xia, "DOIDFH: an Effective Distributed Metadata Management Scheme," in *Proc. Computational Science and its Applications (ICCSA)*, Kuala Lumpur, Malaysia, Aug. 2007, pp. 245 – 252.

[61] Y. Fu, N. Xiao, and E. Zhou, "A Novel Dynamic Metadata Management Scheme for Large Distributed Storage Systems," in *Proc. The 10th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Dalian, China, Sep. 2008, pp. 987 – 992.

[62] "DUNE:Distributed and Unified Numerics Environment."

[63] P. Bastian, M. Blatt, A. Dedner, C. Engwer, and R. Klofkorn, "A generic grid interface for parallel and adaptive scientific computing. Part I: abstract framework," *Computing*, vol. 82, no. 2-3, pp. 103 – 119, 2008.

[64] "OCCI:Open Cloud Computing Interface," Online, http://occi-wg.org/.

[65] R. Nyren, A. Edmonds, and A. Papaspyrou, "Open Cloud Computing Interface Core Specification (GFD-P-R.183)," OCCI-WG, Tech. Rep., 2011, http://ogf.org/documents/GFD.183.pdf.

[66] S. Jha, H. Kaiser, Y. El Khamra, and O. Weidner, "Design and Implementation of Network Performance Aware Applications Using SAGA and Cactus," in *Proc. The 3rd IEEE Conference on eScience2007 and Grid Computing.*, Bangalore, India, Dec. 2007, pp. 143–150.

[67] "The SAGA C++ Reference API," Online, http://saga.cct.lsu.edu/cpp/apidoc/.

[68] "SAGA Middleware Adaptors," Online, http://www.saga-project.org/download/adaptors.

[69] "XML-RPC Specification," Online, http://xmlrpc.scripting.com/spec.html.

[70] "OGF – Open Grid Forum," Online, http://www.ogf.org/.

[71] "RFC2459 – Internet X.509 Public Key Infrastructure Certificate and CRL Profile," Online, http://tools.ietf.org/html/rfc2459.

[72] "Overview of the Grid Security Infrastructure," Online, http://www.globus.org/security/overview.html.

[73] "Apache Derby," Online, http://db.apache.org/derby/.

[74] "FUSE: Filesystem in Userspace," Online, http://fuse.sourceforge.net/.

[75] "SQLite," Online, http://www.sqlite.org/.

[76] "NAREGI Middleware GridVM (Japanese)," Online, http://middleware.naregi.org/Download/Docs/AG-NAREGI-GridVM-j.pdf.

[77] "VOMS: Virtual Organization Membership Service," Online, http://www.globus.org/grid_software/security/voms.php.

[78] "MyProxy, Credential Management Service," Online, http://grid.ncsa.illinois.edu/myproxy/.

[79] "Belle," Online, http://belle.kek.jp.

[80] "ILC – International Linear Collider," Online in Japanese, http://www.linear-collider.org/.

[81] S. Matsuoka, K. Saga, and M. Aoyagi, "Coupled-Simulation e-Science Support in the NAREGI Grid," *Computer*, vol. 41, no. 11, pp. 42–49, 2008.

[82] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva, "Job submission description language (jsdl) (GFD.136)," JSDL-WG, Tech. Rep., 2008, http://www.ogf.org/documents/GFD.136.pdf.

[83] “Geant4,” Online, http://geant4.cern.ch/.

[84] J. Allison *et al.*, “Geant4 developments and applications,” *IEEE Trans. Nucl. Sci.*, vol. 53, pp. 270–278, February 2006.

[85] S. Agostinelli *et al.*, “GEANT4 – A simulation toolkit,” *Nucl. Instrum. Meth.*, vol. A506, pp. 250–303, July 2003.

[86] “RFC1630 – Universal Resource Identifiers in WWW,” Online, http://www.ietf.org/rfc/rfc1630.txt.

[87] “ImageMagick,” Online, http://www.imagemagick.org/script/index.php.

[88] “ImageMagick Program Interfaces,” Online, http://www.imagemagick.org/script/api.php.

[89] “TORQUE Resource Manager,” Online, http://www.clusterresources.com/products/torque-resource-manager.php.

[90] “TORQUE Resource Manager, qsub,” Online, http://www.clusterresources.com/torquedocs21/commands/qsub.shtml.

[91] “Boost.Process,” Online, http://www.netbsd.org/ jmmv/process/.

[92] Y. Kawai, G. Iwai, T. Sasaki, and Y. Watase, “Managing distributed files with RNS in heterogeneous Data Grids,” in *Proc. the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, California, US, May 2011, pp. 494–503, iSBN: 978-0769543956.

[93] L. E. G. Sarmenta, “Sabotage-tolerance mechanisms for volunteer computing systems,” in *Proc. Cluster Computing and the Grid*, Brisbane, Australia, May 2001.

[94] J. Kaczmarek and M. Wrobel, “Modern approaches to file system integrity checking,” in *Proc. The 1st International Conference on Information Technology*, Gdansk, Poland, May 2008.

[95] Z. Yong-Xia and Z. Ge, “MD5 Research,” vol. 2, pp. 271 – 273, Apr. 2010.

[96] J. Dean, “Software Engineering Advice from Building Large-Scale Distributed Systems,” Stanford CS295 class lecture, Tech. Rep., 2007, http://research.google.com/people/jeff/stanford-295-talk.pdf.

[97] “Phil Dykstra’s nuttcp quick start guide,” Online, http://www.wcisd.hpc.mil/nuttcp/Nuttcp-HOWTO.html.

[98] “Digital Imaging and Communications in Medicine (DICOM),” Online, http://medical.nema.org/.

[99] Y. Itow *et al.*, "The JHF-Kamioka neutrino project," *KEK Report*, vol. 4, 2001, 29pp.

[100] "T2K-ND280 collaboration," Online, http://www.nd280.org/.

[101] "Data Intensive Cyber environments (DICE) Center at the University of North Carolina at Chapel Hill," Online, http://dice.unc.edu/.

[102] C. Blum, "Ant Colony Optimization: Introduction and recent trends," *Physics of Life Reviews*, vol. 2, pp. 353 – 373, Oct. 2005.

[103] C. Jiang, C. Wang, X. Liu, and Y. Zhao, "A Survey of Job Scheduling in Grids," *Lecture Notes in Computer Science*, vol. 4505/2007, pp. 419 – 427, 2007.

[104] G. Subashini and M. Bhuvaneswari, "Non Dominated Particle Swarm Optimization For Scheduling Independent Tasks On Heterogeneous Distributed Environments," *Int. J. Advance. Soft Comput. Appl.*, vol. 3 Number 1, Mar. 2011.

[105] A. Abraham, H. Liu, W. Zhang, and T. Chang, "Scheduling Jobs on Computational Grids Using Fuzzy Particle Swarm Algorithm," *Springer-Verlag Berlin Heidelberg*, pp. 500 – 507, 2006.

[106] H. Izakian, B. T. Ladani, K. Zamanifar, and A. Abraham, "A Novel Particle Swarm Optimization Approach for Grid Job Scheduling," *Information Systems, Technology and Management, Communications in Computer and Information Science*, vol. 31, Part 5, pp. 100 – 109, 2009.

[107] A. Abraham, S. Das, and S. Roy, "Swarm intelligence algorithms for data clustering," *In Soft computing for knowledge discovery and data mining*, vol. Part IV, pp. 279 – 313, 2007.

[108] A. N. Sinha, N. Das, and G. Sahoo, "Ant colony based hybrid optimization for data clustering," *Kybernetes*, vol. 36, Issue 2, pp. 175 –191, 2007.

[109] R. Peterson and E. G. Sirer, "Antfarm: Efficient Content Distribution with Managed swarms," *NSDI '09: USENIX Symposium on Networked Systems Design and Implementation*, pp. 107 – 122, 2009.

[110] Y. Yang, Y. Zhao, and F. Hou, "Ant colony optimization algorithm based P2P system replica optimal location strategy," *Service Operations and Logistics, and Informatics*, pp. 494 – 497, Oct 2008.

[111] "Akamai technologies, Globally Distributed Content Delivery," http://www.akamai.com/dl/technical_publications/GloballyDistributedContentDelivery.pdf.

List of Publications

Journals

1. Y. Kawai, A. Hasan, G. Iwai, T. Sasaki, and Y. Watase, "A Swarm Inspired Method for Efficient Data Transfer," Parallel and Distributed Computing and Networking, IEICE, vol. E95-D, no. 12, Dec. 2012, pp. 2852-2859, ISSN: 1877-0509.

Conference Proceedings

1. Y. Kawai, G. Iwai, T. Sasaki, and Y.Watase, "Universal Grid User Interface(UGI) for Multiple Grids and Cloud," in Proc. International Symposium on Grids and Clouds (ISGC), in series Proceedings of Science, Taipei, Taiwan, Mar. 2012.

2. Y. Kawai, A. Hasan, G. Iwai, T. Sasaki, and Y. Watase, "Performance Evaluation of The Software Abstraction Layer (Japanese)," in Proc. the 10th Forum on Information and Technology (FIT), Hakodate, Japan, Sep. 2011, pp. 257-258.

3. Y. Kawai, A. Hasan, G. Iwai, T. Sasaki, and Y. Watase, "A method for reliably managing files with RNS in multi Data Grids," in Proc. International Conference on Computational Science (ICCS), in series Procedia Computer Science, Singapore, Jun. 2011, pp. 412-421, ISSN: 1877-0509.

4. Y. Kawai, T. Sasaki, Y. Iida, Y. Watase, A. Hasan, and F. D. Lodovico, "Managing Large and Small Files in a Distributed System," in Proc. the 5th IEEE International Conference on Digital Ecosystems and Technologies (IEEE-DEST), Daejeon, Korea, Jun. 2011, pp. 182-187, ISBN: 978-1457708718.

5. Y. Kawai, G. Iwai, T. Sasaki, and Y.Watase, "Managing distributed files with RNS in heterogeneous Data Grids," in Proc. the 11th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid), California, US, May 2011, pp. 494-503, ISBN: 978-0769543956.

6. Y. Kawai, G. Iwai, T. Sasaki, and Y.Watase, “SAGA-based application to use resources on different Grids,” in Proc. International Symposium on Grids and Clouds (ISGC), in series Proceedings of Science, Taipei, Taiwan, Mar. 2011.

7. G. Iwai, Y. Kawai, T. Sasaki, and Y. Watase, “A Development of Lightweight Grid Interface,” in Proc. the 18th International Conference on Computing in High Energy and Nuclear Physics (CHEP), Taipei, Taiwan, Jul. 2010.

8. Y. Kawai, G. Iwai, T. Sasaki, and Y. Watase, “An alternative method for reliably managing large files,” in Proc. the 73rd National Convention of Information Processing Society of Japan (IPSJ), Tokyo, Japan, Mar. 2011, pp. 223-224.

9. Y. Kawai, G. Iwai, T. Sasaki, and Y. Watase, “SAGA based Grid Application and the performance evaluation (Japanese),”in Proc. the 9th Forum on Information and Technology (FIT), Fukuoka, Japan, Sep. 2010, pp. 395-399.

10. Y. Kawai, G. Iwai, T. Sasaki, and Y. Watase, “SAGA-based File Access Application over Multi-Filesystem Middleware,”in Proc. Challenges of Large Applications in Distributed Environments (CLADE), in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago, US, Jun. 2010, pp. 622-626, ISBN: 978-1605589428.

11. G. Iwai, Y. Kawai, T. Sasaki, and Y.Watase, “SAGA-based user environment for distributed computing resources: A universal Grid solution over multi-middleware infrastructures,”in Proc. International Conference on Computational Science(ICCS), in series Procedia Computer Science, Amsterdam, Netherlands, May 2010, pp. 1539-1545, ISSN: 1877-0509.

12. Y. Kawai and A. Hasan, “High Availability iRODS System (HAIRS),” in Proc. iRODS User Group Meeting 2010, Chapel Hill, US, Mar. 2010, pp. 11-15, ISBN 1452813426.

13. G. Iwai, Y. Kawai, T. Sasaki, and Y. Watase, “A Prototyping of Web Interface for Treatment Planning in Radiotherapy in the Multi Grid Infrastructure,” in Proc. Asian Simulation Conference (ASC), ShigaCJapan, Oct. 2009.