In: Advanced Parallel and Distributed Computing
Editor: Yuan-Shun Dai, et. al., pp. 141–173

ISBN 1-60021-202-6
© 2006 Nova Science Publishers, Inc.

Chapter 8

GRID PROGRAMMING MODELS AND ENVIRONMENTS

Harold Soh, Shazia Haque, Weili Liao and Rajkumar Buyya[*]
Grid Computing and **D**istributed **S**ystems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia

Abstract

This chapter presents various models for creating Grid applications and runtime environments for managing the execution of applications on global Grids. The chapter discusses superscalar, message passing, remote procedure calls, bag of tasks, distributed objects, threads, workflows, and Grid services programming models supported by existing implementations Gridsuperscalar, MPICH-G, Ninf-G, Nimrod-G/Gridbus Broker, ProActive, Alchemi, Gridbus/Kepler workflow, and Globus Toolkit respectively.

1 Introduction

In the last decade, the world has experienced an explosion in the amount of available data. Businesses, researchers and engineers have gone on a large-scale data harvest, collecting records with tremendous diligence. Due to the high computational complexity involved, the processing of these types of data had been a task exclusive to high-performance computing systems such as supercomputers and clusters. However, the growth in the amount of data and computational time required has outstripped the power afforded by isolated high-performance machines.

In addition, enterprises have grown to a stage where business divisions and units are established at multiple geographical locations nationally and even globally. Each unit has the responsibility of managing their own datasets with some degree of federation. Furthermore, business units have started conducting not only intra-enterprise but also inter-enterprise business, which necessitates the sharing of their information resources and assets with their

[*]Corresponding author: raj@csse.unimelb.edu.au

partners. The same scenario holds true for the scientific community where many of the big-science studies and experiments, such as in the domain of physics and the biological sciences, need to be conducted through global collaboration as no single organization has the capacity or the financial capability to possess all the required resources.

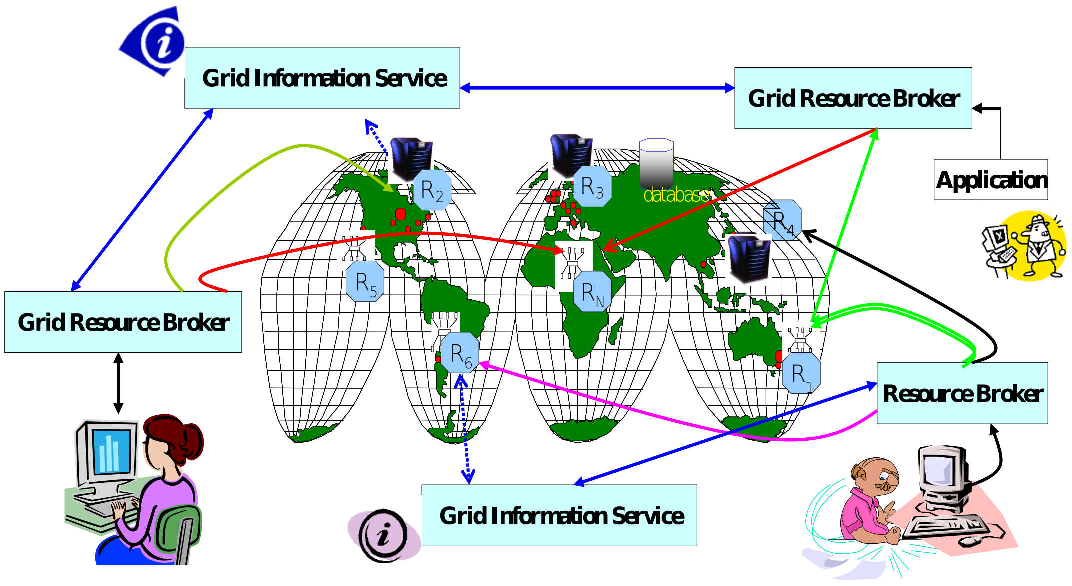


Figure 1: A world-wide Grid computing environment.

Fortunately, the significant increase in the availability of powerful computers and wide-area network performance allows us to combine resources across multiple organizations, yielding the Grid [1]. Grids enable the sharing, exchange, discovery, selection, and aggregation of geographically/Internet-wide distributed heterogeneous resources – such as computers, databases, visualization devices, and scientific instruments [2]. As such, Grids have emerged as the modern cyber infrastructure for the creation of *virtual organizations* (VO) and *virtual enterprises* (VE) [3] [4]. Grids offer us a tremendous computational and data resource, capable of solving many of the worlds most important problems. Examples would include searching for an AIDS vaccine or discovering the secrets of universe through particle physics. Figure 1 illustrates a high-level view of the activities involved within a typical Grid computing environment [2]. Users access the Grid via Grid middleware that perform resource discovery, job scheduling and process monitoring on the Grid resources.

In this chapter, we discuss the different methods of programming applications that will work on Grids. Before we actually delve straight into the subject matter, it would be helpful to consider the various characteristics of Grids that make application development a complex task. Grids are by nature *heterogeneous* and the differences do not start and end with hardware and software. Grid resources are managed by different organizations that may have different policies and procedures which dictate how applications are run on those machines. One or both organizations may decide to take their resources off the Grid at any time, perhaps for upgrading or maintenance. Hence, Grids are heterogeneous in terms of *architecture*, *management systems* and *access interfaces*. The fact that machines of various types can join or leave the Grid at any time and that the interconnections that exist between these machines can change lead us to say that a Grid is *open* and *dynamic*.

Now that we have some idea of what kind of entity the Grid is, we can discuss about what properties an application has to possess in order to successfully function on the Grid. Firstly, it has to be *portable*, meaning it should be able to run on many different types of machines without recompilation. A term we can use to describe this property is *architecture independence*. Performance is another property we have to consider. Often, the main reason for doing work on the Grid is to do it in less time. However, achieving high performance while performing all the background work necessary to ensure correct program execution is not a trivial task. We can say an application should have *performance reliability* meaning it should work efficiently and reliably. Equally important are the properties of *fault tolerance* and *security*. Grid applications should be able to detect and recover from errors resulting from computational and communication faults. In addition, some measure of protection for the data and code running on the various Grid nodes is necessary. Security mechanisms should be present to provide both the client applications and the Grid nodes with safety and privacy.

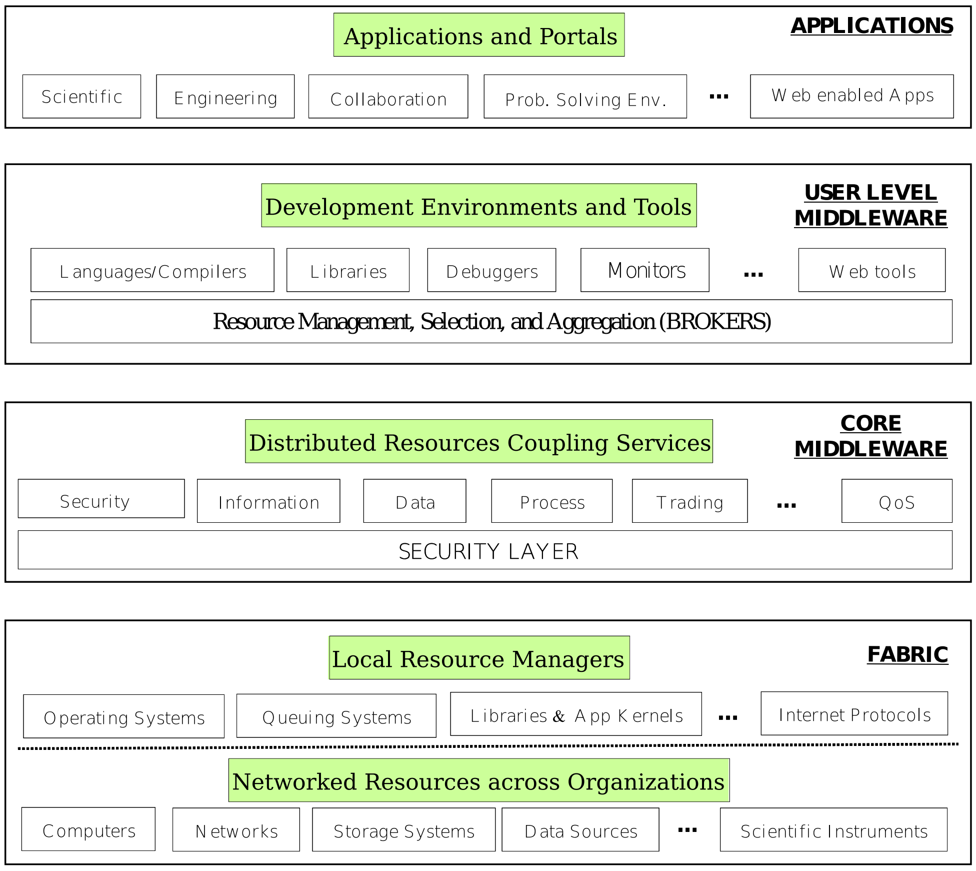


Figure 2: A Layered Grid Architecture and Components.

Designing an application to possess all the aforementioned properties is likely to be complex and difficult. Research over the past two decades has yielded some interesting models and environments that simplify Grid programming, making it accessible to mainstream developers. Figure 2 illustrates the various layers within typical Grid architecture. Grid development or programming environments sit right below the applications layer, providing an abstraction of the services offered by the core middleware layer. The core middleware interacts with the Grid fabric which consists of the lower-level software and hardware components that make up the Grid, such as local resource managers, individual operating systems, computers, networks and communication protocols.

A popular and widely used middleware is the Globus toolkit [5], which provides a number of services including communication, resource management, security services and file access services. The Globus Toolkit provides five main types of services as outlined below:

1. **Communication:** Multi-method communication is supported via the Nexus communication library.

2. **Resource Management:** The Globus Access to Secondary Storage (GASS) is a file access mechanism that allows applications to pre-fetch and open remote files and write them back. GASS is generally used for executable and input file staging and for relaying output back upon completion. The Globus Resource Allocation Manager (GRAM) provides for remote execution and status monitoring.

3. **Information Services:** The Monitoring and Discovery Service (MDS) provides information about the Grid nodes.

4. **Data Management:** The GridFTP and Replica Location and Management components provide utilities and libraries for transmitting, storing and managing large sets of data.

5. **Security:** The Grid Security Infrastructure (GSI) provides authentication, authorization and secure communication services via single sign-on and data encryption.

2 GRID PROGRAMMING APPROACHES

Grid-enabling applications involves two major undertakings; *program decomposition* (or *task composition*), *resource composition*. The Grid programming environments described in this chapter simplify Grid application programming by automating either (or both) of these tasks to a certain degree. It should be noted that these tasks can (and are likely to) be interdependent. Program decomposition may rely on what resources are available and the set of resources composed may differ depending on the application requirements.

2.1 Resource Composition

Resource composition is a two step process. First, it is necessary to perform resource discovery, the identification of Grid resources that are available for use. Resources may be listed in an accessible Grid directory service or a default list can be provided by the user. Next, the proper resources need to be selected based on the application processing/data requirements and additional external constraints such as financial cost or execution deadline.

Resource composition can be performed by an appropriate application component or by a Grid resource broker such as the Gridbus broker [6] or Nimrod-G [7].

2.2 Program Decomposition

Decomposing a program involves splitting the work (program code or data) into chunks that can be distributed to Grid resources for proper handling. Taking a very simplistic view, we can classify program decomposition into three approaches, *implicit*, *explicit* and *semi-implicit*.

With implicit parallelism, programs are automatically parallelized by the environment and it is not necessary to identify sections of code that can be performed in parallel, scheduling or data dependencies. An example of an environment capable of this is Grid superscalar [8], which we discuss later in this chapter. On the other end of the spectrum, explicit parallelism requires the programmer to be responsible for most of the parallelization effort such as task decomposition, mapping tasks to processors and inter-task communication. Examples of these programming approaches include the explicit communication models such as message passing and remote procedure calls.

It is not difficult to recognize that both the implicit and explicit parallelism approaches have their advantages and disadvantages. Implicit parallelism allows us to rapidly develop Grid applications but we lose fine grain control. Explicit parallelism gives us near complete control of how our applications will execute but in doing so, cost us time and effort. Choosing one approach over the other depends largely on the task at hand. Several models have opted for a balance, a middle ground that offers a good deal of program control while automating some of the Grid management details. We call these semi-implicit parallelism models and examples would include the bag of tasks, distributed threads and workflow models.

2.3 Grid Programming Models

The table below summarizes several different grid programming models and environments:

In the following Sections 3 to 10, we present an overview of each of these programming models with associated environments to illustrate the principles behind each model.

3 GRID SUPERSCALAR

The first programming model and environment we will be examining is Grid superscalar which is undergoing further development at the CEPBA-IBM Research Institute in Spain. Recall from our earlier discussion that Grid superscalar is an implicit parallelization Grid programming environment. The underlying idea is that Grid applications consist of repetitive tasks which can be detected and parallelized automatically while guaranteeing correct program execution. Hence, the programmer is only required to provide two files, the sequential source code in an imperative language (C/C++ or Perl) and an interface definition (IDL) file in the CORBA IDL language [16]. The IDL file specifies the subroutines/programs that are to be executed on the Grid and parameters (input/output files or

Table 1: Summary of Grid Programming Models and Environments.

Model	Environment	Description
Superscalar	**Grid Supercalar** *http://people.ac.upc.edu/ rosab/index_gs.htm*	Superscalar is a common concept in parallel computing. Sequential applications composed of tasks of a certain granularity are automatically converted into a parallel application where the tasks are executed in different servers of a computational Grid [8].
Explicit Communication **(Message Passing, Grid Remote Procedure Call)**	**MPICH-G2** *http://www3.niu.edu/mpi/*	MPICH-G2 is a Grid-enabled implementation of the Message Passing Interface (MPI) [9]. MPI defines standard functions for communication between processes and groups of processes.MPICH-G2 provides extensions to MPICH using the Globus Toolkit, giving users familiar with MPI an easy way of Grid-enabling their MPI applications.
	Grid–enabled RPC *http://ninf.apGrid.org/*	GridRPC is a Remote Procedure Call (RPC) model and API for Grids [10]. Besides providing standard RPC semantics, it offers a convenient, high-level abstraction whereby many interactions with a Grid environment can be hidden.
Bag of Tasks	**Nimrod-G** *http://www.csse.monash.edu. au/∼davida/nimrod/nimrodg .htm*	The Nimrod-G Broker [7] is a Grid-aware version of Nimrod, a specialized parametric modeling system. Nimrod uses a simple declarative parametric modeling language and automates the task of formulating, running, monitoring, and aggregating results.
	Gridbus Broker *http://www.Gridbus.org/ broker*	The Gridbus Broker is a software resource that permits users access to heterogeneous Grid resources transparently [11]. Gridbus Broker Application Program Interface (API) provides a straightforward means to users to Grid-enable their applications with minimal extra programming [12].
Distributed Objects	**ProActive** *http://www-sop.inria.fr/oasis/Proactive/*	ProActive is a Java based library that provides an API for the creation, execution and management of distributed active objects. Proactive is composed of only standard Java classes and requires no changes to the Java Virtual Machine (JVM) allowing Grid applications to be developed using standard Java code.
Distributed Threads	**Alchemi** *http://www.alchemi.net*	Alchemi is a Microsoft .NET Grid computing framework, consisting of service-oriented middleware and an application program interface (API) [13]. Alchemi features a simple and familiar multithreaded programming model.
	Grid Thread Programming Environment (GTPE)	GTPE is a programming environment implemented in Java utilizing the Gridbus Broker API. GTPE further abstracts the task of Grid application development, automating Grid management while providing a finer level of logical program control through the use of distributed threads.

Workflow	**Kepler** *http://kepler-project.org*	Kepler is a scientific workflow management system along with a set of Application Program Interfaces (APIs) for heterogeneous hierarchical modeling [14]. Kepler provides a modular, activity oriented programming environment, with an intuitive GUI to build complex scientific workflows.
Grid Services	**OGSA** *http://www.ggf.org/ ggf_areas_architecture.htm*	Open Grid Services Architecture (OGSA) [3] [15] is an ongoing project that aims to enable interoperability between heterogeneous resources by aligning Grid technologies with established Web services technology. The concept of a *Grid service* is introduced as a Web service that provides a set of well defined interfaces that follow specific conventions. These Grid services can be composed into more sophisticated services to meet the needs of users.

generic scalars). From these two files, the system generates a parallel application where tasks are executed on the Grid.

3.1 Automatic Code Generation

Grid superscalar offers a tool **gsstubgen** that generates parallel code for the user automatically. This automatically generated code consists of two files: the function stubs and the skeleton for the code that is to be executed on the remote servers. Figure 3 [8] illustrates an example of how files are linked to obtain the final application binaries. One executable will exist in the client host and one in each server host. The original main program (app.c) is linked with the generated stubs (app-stubs.c) on the client machine. On any one server being utilized, the skeleton (app-worker.c) is linked with the file containing the code of the original user functions (app-functions.c) [8].

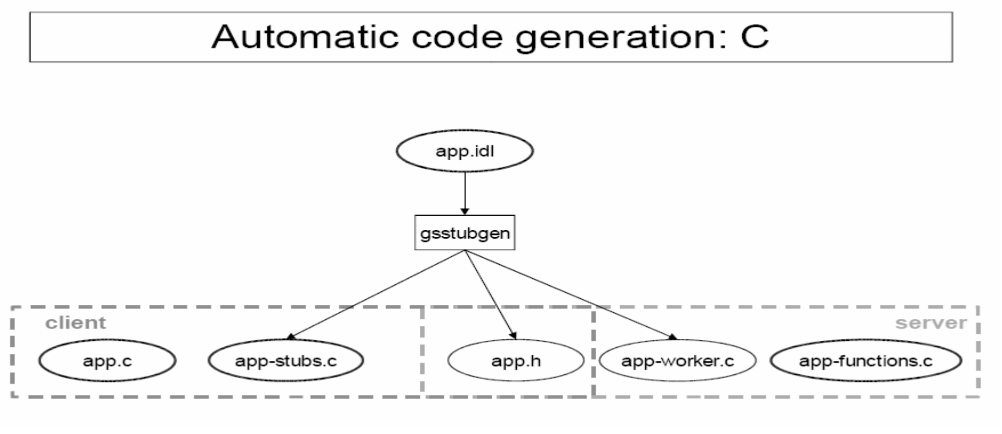
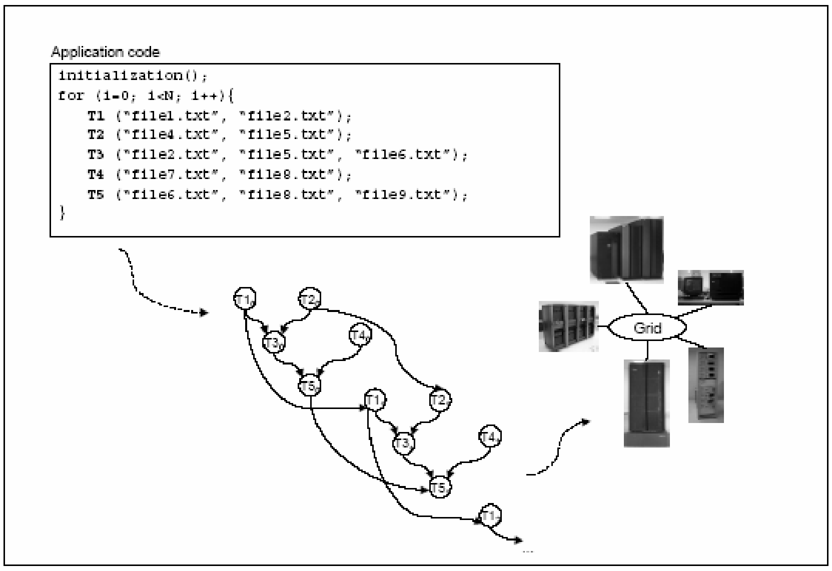


Figure 3: Automatic code generation (C program) [8].

3.2 Run Time

Figure 4 [8] illustrates an instance of Grid superscalar behaviour. The GRID superscalar run-time system looks for data dependencies which is analysed from the input/output files between the different tasks. Tasks are denoted as nodes or vertices on the task dependence graph and data dependencies as edges. Intuitively, tasks that are not connected do not depend on each other and hence, can be executed in parallel.


Application code

```
initialization();
for (i=0; i<N; i++){
    T1 ("file1.txt", "file2.txt");
    T2 ("file4.txt", "file5.txt");
    T3 ("file2.txt", "file5.txt", "file6.txt");
    T4 ("file7.txt", "file8.txt");
    T5 ("file6.txt", "file8.txt", "file9.txt");
}
```

Figure 4: Overview of Grid superscalar behaviour [8].

1. Read after Write (RaW) occurs when a task reads a parameter that is written by a previous task. For example, consider the situation where *functionB* reads some file, *file1*, which was written to by *functionA*. In this case, the run time will ensure that *functionB* is executed after *functionA*.

2. Write after Read (WaR) occurs when a task writes to a parameter that is read by a previous one. For this one, we consider the reverse of the situation we examined earlier. If some *functionA* writes to *file1* after it is read by *functionB*, then *functionA* should be executed after *functionB* to prevent what we call *dirty reads*.

3. Write after Write (WaW) occurs when a task writes to a parameter which is also written to by a previous task. Let us assume that *functionA* writes to *file1* after *functionB* and there exists a *functionC* reads *file1*. The order in which *functionA* and *functionB* get executed will affect the information which *functionC* reads.

It is possible to eliminate both WaR and WaW dependences through proper renaming of parameters. Grid superscalar run time does this automatically via the use of a hash table. However, it is not possible to eliminate RaW dependences and so, we say RaW are *true dependences*.

3.3 Task Submission and End of Task Notification

If a task does not have any dependence on previous tasks which have not been finished or which are still running (i.e. the task is not waiting for any data that has not been already generated), it can be submitted for execution to the Grid. The Grid superscalar run-time requests a Grid server from a resource broker and if a server is provided, it submits the task for execution. Currently, Grid superscalar is packaged with a simple resource broker but future work includes interfaces to more sophisticated resource brokers such as the Gridbus broker which we will examine later in this chapter. Task submission consists of two steps:

1. File submission whereby input files are transmitted to the Grid servers.

2. Task submission whereby the task itself is called to be executed on the Grid server.

When a task completes execution, it notifies the superscalar run-time so that tasks that depend on this task and have no other dependencies can be submitted for execution. This process of task submission proceeds until the program terminates.

4 MESSAGE PASSING

The concept of message passing is a common parallel programming paradigm. The Message Passing Interface (MPI) [17] [18] is widely used in cluster applications. MPI applications do not share memory but exchange information via messages over some medium, such as an Ethernet network. We say that the processes in an MPI application run in *disjoint address spaces*. Explicit parallelization via message passing is cumbersome compared to the automatic parallelization offered by implicit techniques like Grid superscalar. However, the flexibility and control gained may be necessary for certain applications where automatic parallelization fails.

4.1 The MPI Standard

In this subsection, we give an overview of the major features and design of MPI. The MPI Standard describes a standard message passing interface for distributed machines. At the simplest level, MPI provides a reliable communication mechanism for sending and receiving messages.

The most basic point-to-point communication operations are the send and receive operations which can either be *blocking* or *non-blocking*. Blocking sends will not return until the data locations specified in the message can be used without corrupting the message. Likewise, blocking receives do not return until the message has been successfully copied into the data block specified. Non-blocking sends and receives return immediately, not waiting for any particular event. In addition to the data being sent, messages contain a fixed number of fields which are collectively called the message envelope. This message envelope (consisting of the source, destination, tag and communicator) distinguishes messages and allows for message selectivity.

4.1.1 Process Groups

It is possible to organize senders and receivers into process groups. Each process group is an ordered collection of processes with each process uniquely identified by its rank. The rank identifier for a process group is from 0 to n-1 for a process group of size n. Although the number of processes is static or fixed for the lifetime of a MPI program, process groups are dynamic. Process groups can be created, destroyed and the same process can belong to multiple process groups. Using process groups, MPI nodes can be structured for collective communication and to enable task parallelism.

4.1.2 Communication Objects and Contexts

Communicators are abstract objects that define a scope of a communication operation, which is defined by the process groups involved and the communication context. Communication contexts, like message tags, groups and rank identifiers, provide a mechanism for message selection. Communication contexts are maintained transparently within communicators so that messages sent through a particular communicator can only be received through the matching communicator. These attributes provide for both point-to-point and collective communication.

4.2 MPICH-G2

MPI is a message passing interface and not a full-fledged or complete Grid programming environment. As such, MPI does not contain inherent support for fault tolerance, file sharing and distribution or security mechanisms. As discussed earlier, these services are essential for correct and efficient program execution on a Grid. MPICH-G2 [9] is based on the MPI-1 standard and was developed to enable users to run MPI programs on the Grid without changing the standard commands. MPICH-G2 was constructed using two software systems, MPICH [19] and the Globus middleware toolkit. The following services are provided by MPICH-G2 system:

1. **Co-allocation:** The co-allocation problem involves the allocation of resources on Grid nodes, the initialization of processes and mechanisms to link these processes for communication. Complications arise because of two main factors:

   (a) Heterogeneity: Different nodes may differ in the methods utilized for resource allocation and process creation.

   (b) Errors: Co-allocation can be time-intensive and error prone due to the dynamic nature of the Grid.

   MPICH-G2 solves these issues by using the GRAM interface provided by the Globus toolkit and the Dynamically-Updated Request Online Co-allocator (DUROC). DUROC handles request submission, startup verification and process linking under an umbrella communicator, MPI_COMM_WORLD, which spans all processes.

2. **Security:** MPICH-G2 supports authentication and authorization via the Globus Security Infrastructure (GSI) which provides single sign-on and automatic mapping to local accounts.

3. **Executable staging and results collection:** Once the proper security protocols have been observed, executables are transferred to remote nodes via the Globus Access Secondary Storage (GASS) service. GASS is also used to collect standard output and error streams.

4. **Communication:** The Nexus library provided by the Globus toolkit is used to link processes together allowing them to communicate via TCP/IP in the wide area, shared memory within a cluster and vendor specific protocols within a cluster.

5. **Monitoring:** The GRAM callback functions are used to detect process termination and the GRAM API control functions allow for the termination of processes.

5 REMOTE PROCEDURE CALLS

This section gives an overview of the GridRPC standard [10] [20] and the Ninf-G GridRPC implementation [21]. Remote Procedure Calls (RPC) methods are not too different from the message passing concept discussed in the previous section. However, instead of sending messages through the use of various arguments to a library call, interaction is based on function calls. As such, communication between distributed processes is more of a language construct. One of the key benefits is that the receiver does not have to directly interpret messages.

5.1 GridRPC

GridRPC seeks to combine the standard RPC programming model with asynchronous course-grained parallel tasking. In the GridRPC model, there exist three major types of entities as illustrated by Figure 5 (adapted from [20]); the client, service and registry.

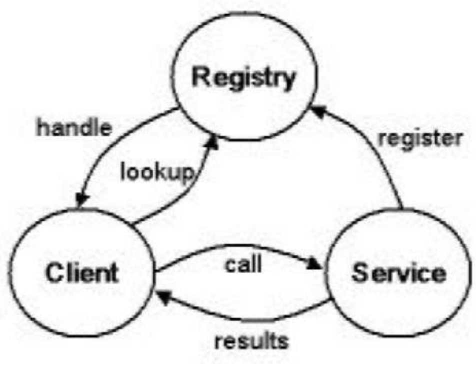


Figure 5: The Basic GridRPC model. Adapted from [20].

Clients make use of services which have registered themselves with a proper registry. When an RPC client performs a look-up for a desired service, the registry returns a *function handle*. The function handle represents a mapping from a flat function name string to an instance of that function on a Grid node. The client then can make a RPC call using that function handle to execute the function, which returns results after it completes. Another term we have to be familiar with is the *session ID* which is an identifier representing a specific *non-blocking* GridRPC call. The GridRPC API provides multiple data types and methods for function initialization, creation and destruction of function handles, function calls, asynchronous waits and error reporting. We direct readers wanting a full description of these API functions to the GridRPC model and API document [15].

5.2 Ninf-G GridRPC system

Ninf-G was designed by researchers at the National Institute of Advanced Industrial Science and Technology and the Tokyo Institute of Technology to simplify the development of large-scale Grid programs. Ninf-G was built as a GridRPC layer on top of the Globus Toolkit as illustrated in Figure 6 [21] and utilizes GRAM to invoke remote executables, MDS to publish internation information and file paths of components, Globus I/O for communication and GASS for results and error collection.

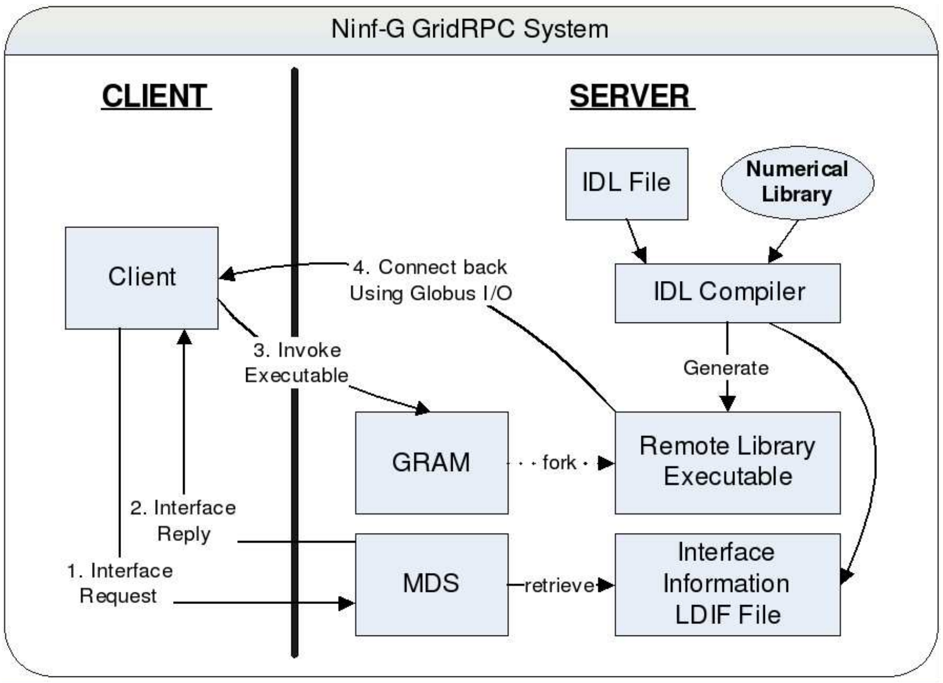


Figure 6: The Ninf-G GridRPC System.

The Ninf-G client APIs provide the following functions:

1. **Initializing and Finalizing functions** similar to that of MPI.
2. **Function Handle Management functions** that allow for the creation and destruction of function handles.
3. **GridRPC Call functions** that can be blocking or non-blocking and may use a variable number of arguments or an argument stack.

4. **Asynchronous GridRPC Control Functions** that are used to probe or terminate outstanding non-blocking function calls.

5. **Asynchronous GridRPC Wait functions** that are used to wait instead of polling on a set of session IDs.

6. **Error Reporting functions** that provide error codes and human-readable error descriptions in the event on an error.

7. **Argument Stack functions** such as *push* and *pop* that allow for the run-time construction of arguments.

5.2.1 Server-Side Library Interface Information

The Ninf Interface Description Language (IDL) is used to specify interface information for Grid libraries hosted on servers. The IDL file specifies the Grid functions that can be called, which arguments are input or output, argument types as well as information needed to compile and link the necessary libraries. The IDL files can be compiled into stub main routines and makefiles. To provide access to libraries or applications over the Grid using Ninf-G, four main steps are required:

1. Create an IDL interface file for the library function or application.

2. Compile the IDL file and generate a stub main routine and a makefile for the remote program.

3. Compile the stub main routine and link it with the remote library.

4. Publish the necessary information (via MDS).

The final two steps are automatically performed by the makefile and no IDL handling is required on the client.

5.2.2 Utilizing GridRPC

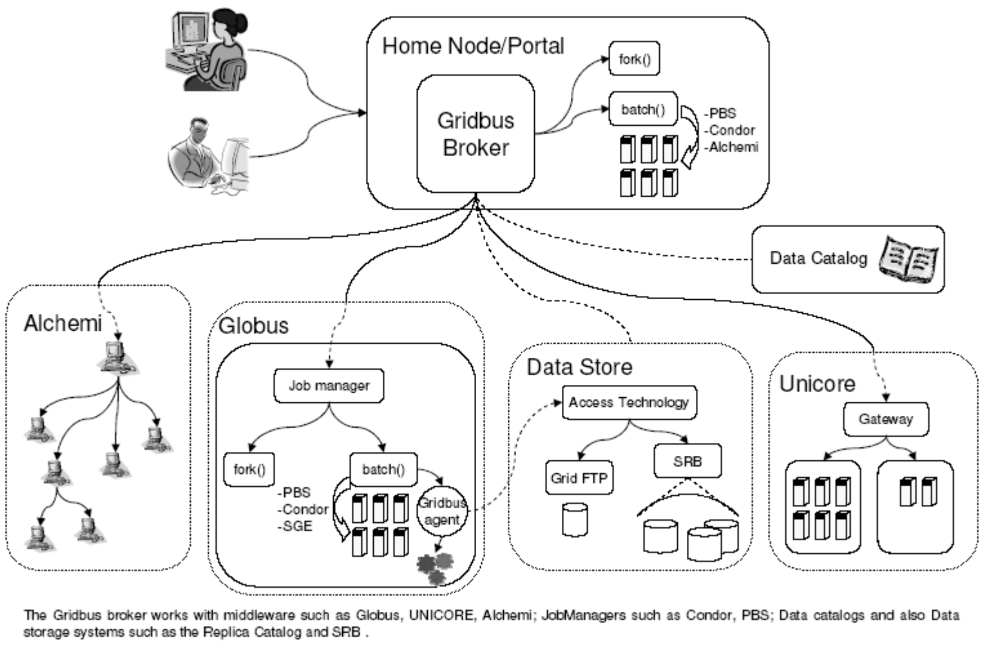
Invoking the relevant functions published by the remote libraries involves three main steps. The client performs a query to the MDS and obtains the interface information along with an executable pathname that was registered. Following this, the client and server mutually authenticate each other using GSI and the client invokes the remote executable. Finally, the remote executable callbacks to the client utilizing the Globus I/O for further communication (e.g. parameter transfer and error reporting).

6 BAG OF TASKS

The Bag of Tasks (BoT) paradigm involves treating applications as being composed of independent tasks that can be performed in parallel. An example of a BoT model is the parameters sweep, which distributes the same program across multiple Grid nodes to work on different parameters. Such processes may require the services of shared resources such as databases or authentication servers. Examples of a successfully developed parameter sweep applications include a molecular modeling application using the Nimrod-G resource broker [7] for drug discovery [22] and a high energy physics (HEP) application modeling the decay of B-mesons using the Gridbus Broker [6] [7]. The following subsection details the Gridbus Broker but similar principles underly the Nimrod-G broker. We direct readers wanting more information regarding Nimrod-G to [7].

6.1 Gridbus Broker

The Gridbus Broker [11] [12] is a software resource that allows users to access heterogeneous Grid resources transparently. Implemented in Java, it provides a variety of services including resource discovery, transparent access to computational resources, job scheduling and job monitoring. The Gridbus broker transforms user requirements into a set of jobs that are scheduled on the appropriate resources, managing them and collecting results. Figure 7 [12] illustrates the possible interactions the Gridbus broker can participate in.



The Gridbus broker works with middleware such as Globus, UNICORE, Alchemi; JobManagers such as Condor, PBS; Data catalogs and also Data storage systems such as the Replica Catalog and SRB .

Figure 7: Gridbus broker block diagram [12].

6.1.1 Design

The Gridbus broker was designed based on object-oriented principles to be simple, modular, reusable, extendable and flexible. There are six main design entities within the Gridbus broker:

1. **Compute Server:** This entity describes a computational node on the Grid, specifying relevant properties such as middleware, architecture and operating system. The entity also implements a monitor for tracking the rate of progress through the number of jobs that have finished, failed or are currently executing.

2. **Job:** A job is an abstraction for a unit of work submitted to a Grid node for execution. A job consists of *variables* and a *task*. Variables specify the parameters associated with the job. The task is a description of what is to be done on the compute server and consists of *commands*. A command is one of either three types; Copy, Execute or Substitute. The Copy Command and Execute Command are fairly self-explanatory. The Copy Command copies a file from the source to a remote node (and vice versa) and the Execute Command executes a specified program on the remote node. The Substitute command tells the broker to substitute values for variable names in text files, automating the generation of configuration files for each job.

3. **Data Hosts:** Data hosts describe nodes that contain data files with specifics such as file access protocols and file paths.

4. **Data Files:** Data Files link to the Data Hosts that store the file and specify properties of input files such as size and location.

5. **Farming Engine:** The farming engine is the central component that maintains the overall state of the broker. It contains all the job and server collections and interacts with external applications.

6. **Scheduler:** The scheduler is responsible for distributing jobs to Grid nodes. It is middleware independent and is capable of scheduling jobs based on metrics that are not platform-dependant.

6.1.2 Architecture

The broker consists of three main sub-systems; the Application Interface, the Core and the Execution sub-systems. A high level overview of the three subsystems is shown in Table 2. Figure 8 [12] illustrates the Gridbus broker architecture.

6.1.3 Grid Programming with the Gridbus Broker

The Gridbus broker provides an Application Program Interface (API) that allows users to program the broker and use its services in a user-developed Java application. Developing a Java application that utilizes the broker is relatively simple. It is first necessary to create an instance of **GridFarmingEngine**. The broker's properties can be configured via the broker configuration file (Broker.properties). If no configuration file is found, default values are used. A listing of default values and their properties can be found in the Gridbus broker manual [12]. It is then necessary to set up jobs and servers. There are two methods of achieving this:

Table 2: The three main sub-systems of the Gridbus Broker.

Sub-system	**Description**
Application Interface	• Accepts input to the broker consisting of an application-description (tasks and associated parameters with values) and a resource description.
Core	• Converts application-descriptions to job entities. • Converts resource-descriptions into server entities, which represent grid nodes. • Evaluates task and data requirements to discover appropriate resources. • Schedules jobs and submits using the execution sub-system. • Updates the book-keeper using the job monitoring component of the execution sub-system.
Execution	• Interacts with the Scheduler. • Submits jobs to the remote grid via the actuator component. • Provides job monitoring services.

1. The simpler method is to create an application-description file[1] and a resource list file and provide these filenames to the Farming engine.

2. The more flexible method of using the **Task**, **Command** and **ServerFactory** APIs. Details on using these APIs follow.

To use the API method, first instantiate a new **Job** object and call **Job.setJobID()** to set the job's identification. Then set up the commands that need to be run. The commands objects which are available are the copy commands (**CopyCommand**, **MCopy**[2] and **GCopy**), the **ExecuteCommand**, and **SubstituteCommand**. All three types provide methods that allow the user to set member variables necessary for proper execution (e.g. **CopyCommand.setSource()** and **CopyCommand.setDestination()**). These commands are then added to a created **Task** object via **Task.addCommand()**. After all the commands have been added, the task is provided to the **Job** object using **Job.setTask()**. Variables can be

[1]Application-description files are written in XPML format.

[2]Instructs the broker to copy multiple files using wildcards. More information available in the broker API at http://www.gridbus.org/broker/2.0/docs/.

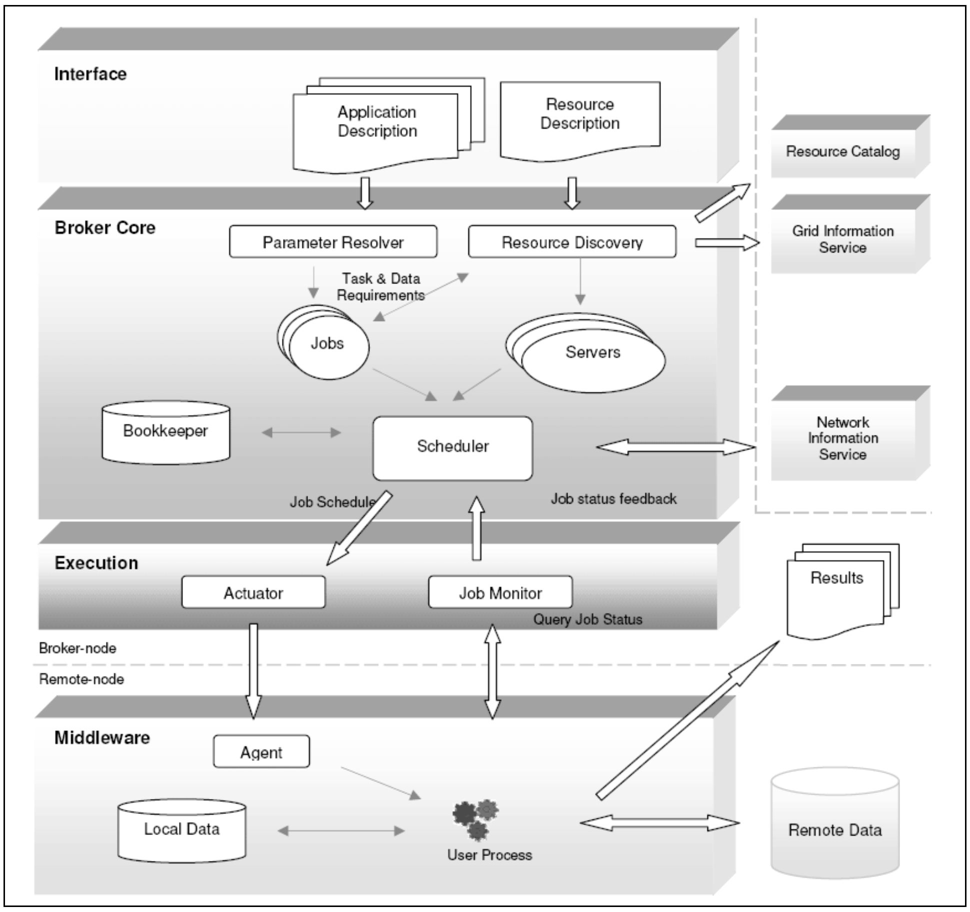


Figure 8: Gridbus broker Architecture.

added to the **Job** object using **Job.addVariable()**. Finally, the job is added supplied to the **FarmingEngine.addJob()** method.

Server resources are supplied to the farming engine, **GridFarmingEngine.addServer()**, as **ComputeServer** objects. A **ComputeServer** object can be generated via **ServerFactory.getComputeServer()** method (e.g. **ComputeServer** cs **= ServerFactory.getComputeServer(** "globus2.4" **,** "belle.cs.mu.oz.au"**)**). We can now schedule the jobs using GridFarmingEngine.schedule(). It is possible to set up the scheduling method or even define a new scheduler by using the **GridFarmingEngine.setScheduler()** method.

The Gridbus broker also provides an API for modifying the broker but this is outside the scope of this chapter and is detailed in [12].

7 DISTRIBUTED OBJECTS

Most readers should be familiar with object oriented programming. Object oriented programming or OOP is a widely used computer programming paradigm where computer programs are viewed as collections of individual units called *objects*. Objects are run-time entities instantiated from classes which encapsulate data and functions. Java is an example of an object oriented programming language. Examples of object-oriented programming languages include Java and C++. Distributed objects are objects that are distributed across multiple computing systems, communicating via messages across some communication network. In the context of Grids, these objects may be located across multiple organizations connected via the Internet.

7.1 ProActive

ProActive [23] extends Java with a Grid API library for the creation, execution and management of active distributed objects with the intention of simplifying parallel computing on LANs, clusters and Internet Grids. ProActive is composed of only standard Java classes and requires no changes to the Java Virtual Machine (JVM) allowing Grid applications to be developed using standard Java code. The ProActive library is based on an Active Object Pattern which is a standard method of encapsulating a remote object, a thread, an actor, a server and a secure mobile entity. In addition, ProActive features group communication, object oriented Single Program Multiple Data (OO SPMD), distributed and hierarchical components, security, fault tolerance, a peer-to-peer infrastructure, a graphical user interface and a powerful XML-based deployment model.

7.1.1 Active Objects

In standard Java, existing code has to be extensively modified to transform local objects into distributed objects, presenting a barrier to developers. ProActive provides simple methods of transforming standard objects into Active objects which possess synchronization capabilities and location and activity transparency. Grid Applications are structured into subsystems, each of which is composed of a single active object and a number of passive objects. Each active object consist of Passive objects are not shared between subsystems. Active objects are composed of two objects, namely a *body* and a standard Java object, and can be created on any host involved in the activity. The body receives stores and executes calls (requests) made to the object. Calls are stored in a queue of id no synchronization policy id provided, manages them in a first-in-first-out (FIFO) manner. The authors note that no parallelism is provided inside of an active object.

7.1.2 Migration and Group Communication

Any active object is capable of migration, which is either self-triggered or initiated by an external agent. All referenced passive objects would also be migrated. Migration relies on serialization and hence, all active objects implement the serializable interface. ProActive implements a simple scheme for enabling group communication. Groups can be easily created and method calls to a group of objects are broadcasted to all members by default. However, it is also possible scatter parameters through the use of the member's rank in a group. Additionally, group communication can be used to simulate MPI-style collective communication within the OO SPMD programming model.

7.1.3 Security

ProActive provides a set of security features that can be used transparently by applications. These features include communications authentication, integrity and confidentiality, migration security, hierarchical security policies and dynamic policy negotiation. The security framework allows for the dynamic deployment of applications and the automatic configuration of security in accordance with the deployment. Each active object has a distinguished element, termed the *root*, which is the only entry point. As such, all other objects within the subsystem are passive objects and cannot be referenced directly. Access is based on the Public Key Infrastructure (PKI) with each entity possessing its' own certificate and a private key generated from the certificate of the user. Additionally, ProActive allows for all RMI and HTTP communication to be tunneled through SSH. As such, all communication can be encrypted and firewalls blocking RMI ports can be bypassed.

7.1.4 Fault Tolerance

ProActive provides a fault-tolerance through a fully-transparent Communication-Induced Checkpointing protocol. Active objects are made persistent through the use of serialization and hence, an object checkpoint consists of a serialized copy of the object and protocol-related information. Each persistent object has to checkpoint at least every TTC (Time to Checkpoint) seconds. A global state is formed when all objects have been checkpointed. In the event of a failure, the system restarts from the global checkpoint. The TTC value is user-defined and can be set to balance the overhead associated with frequent check-pointing and the smaller roll-back time associated with more recent global states.

7.1.5 Web Services Functionality

Active objects can be exported as web services and as such, can be called from any web service language including C#. A web service is a software entity that can be exposed, discovered and accessed by heterogeneous resources over a network in a standard way. ProActive utilizes the SOAP Engine and HTTP servers to enable this functionality.

8 DISTRIBUTED THREADS

Thread programming is a well developed model and is used extensively, even on single processor machines to simplify application development. One can think of threads as light weight processes. For example, a single program could consist of two threads: one to manage the graphical user interface (GUI) and another to perform the actual computations. Distributed threads are threads that span multiple address spaces [24]. In this section, we discuss two programming environments, Alchemi [8] [20] and the GridThread Programming Environment (GTPE) [25], that utilize the distributed thread model.

8.1 Alchemi

Alchemi [13] [26] is Microsoft .NET Grid computing framework, consisting of service-oriented middleware and an application program interface (API) geared towards simple, rapid Grid software development. Alchemi is based on the master-worker parallel programming paradigm and implements the concept of Grid threads. A Grid thread is essentially a thread object capable of running on distributed nodes and is the smallest unit of parallel execution. Hence, an Alchemi Grid application consists of multiple Grid threads.

8.1.1 Owner, Manager, Executor and Cross-Platform Manager

Alchemi consists of four major distributed components described in Table 3. Figure 9 [26] illustrates the interactions between the components.

Table 3: The four main components of the Alchemi Framework.

Component	**Description**
Owner	Executes applications created with the Alchemi API. Submits threads to the Manager and collects completed threads.
Manager	Schedules and manages the execution of threads on executors. Tracks of the availability of executors.
Executor	Accepts and executes threads from the Manager. Can be configured to be dedicated or non-dedicated (Dedicated Executors expose an interface so that the Manager may communicate with it directly. Non-dedicated Executors perform work on a voluntary basis and poll the Manager for threads to execute).
Cross-Platform Manager	A sub-component of the Manager. A web-services interface that enables Alchemi to manage the execution of platform independent grid jobs.

Grid thread scheduling is performed by the Manager and is performed on a Priority and First Come First Served (FCFS) basis. Priorities can be specified when threads created within the Owner (defaults to highest priority if none is specified). Alchemi can be deployed as a hierarchical multi-cluster system with one Manager passing threads to another Manager (defined as the Intermediate-Manager) for execution. As an Intermediate-Manager receives a thread from higher-level Managers, the thread's priority level is reduced by one unit. This serves to allow resources within one administrative domain (managed by a Manager) to be shared without creating a significant impact to local users.

8.1.2 Grid Application Programming with Alchemi

Alchemi[3] provides a .NET Software Development Kit (SDK) consisting of standard classes and an API. An Alchemi Grid application consists of two parts:

[3]A good tutorial on programming with Alchemi can be found at the Alchemi Documentation at http://www.alchemi.net/doc/0_6_1/index.html.

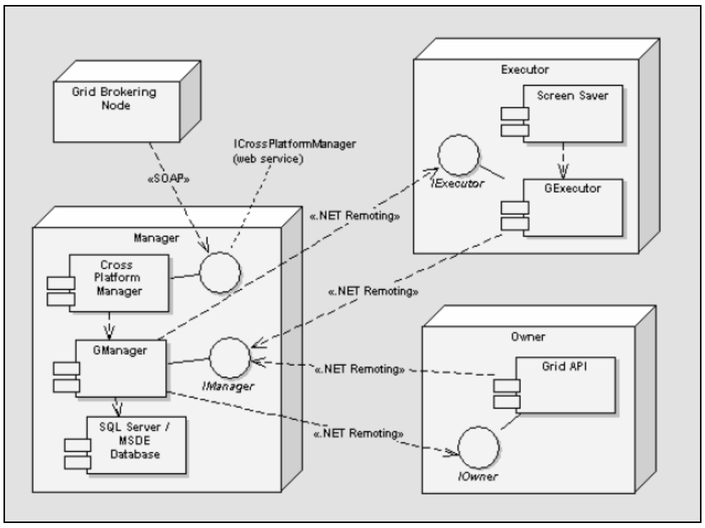


Figure 9: The interactions between the four main components of the Alchemi Framework.

1. "Local Code" which creates a Grid application and runs threads and

2. "Grid Code" which is executed remotely i.e. a Grid thread class[4].

To create a Grid thread class, derive a new class from the **Alchemi.Core.GThread** class. It is necessary to override the **void Start()** method and add the **Serializable** attribute to the class. Then additional modifications (e.g. addition of member functions etc.) can be made to the derived class to perform the necessary computations.

The "Local Code" portion executes on the owner and can be implemented in a variety of ways. The standard method is to create a **GApplication** object with the host and port number provided to the constructor. It is then necessary to create a **ModuleDependency** object and provide it with the module of the derived Grid thread class. The **ModuleDependency** object is added to the **Manifest** object within the **GApplication** object (i.e. **GApplication.Manifest**) via the **Manifest.Add()** method.

The derived threads are then instantiated. A **GThreadFinish** delegate which is called when the thread finishes executing is set for each thread (e.g. **myThread.FinishCallback = new GThreadFinish(ThreadFinished)**). The **ThreadFinished** function, as used in the example, is user-defined and may consist of code to perform cleanup operations or to save the results to disk. It is possible to define a delegate for each thread created. The threads are then added to the **GApplication** object via the **GApplcation.Threads.Add()** method.

As with the threads, we can set an application callback method by creating a **GApplicationFinish** object with the function passed to the constructor and setting the **Gapplication.FinishCallback** member variable. The application is then ready to be started via the **GApplication.Start()** method.

[4]Inter-thread communication is currently not supported by the version of Alchemi (v.0.61) at the time of writing. It is then necessary to break the application into parallel threads that do not require constant communication with each other.

8.2 Grid Thread Programming Environment (GTPE)

The main objective of implementing GridThreads library for Gridbus Broker was to minimize the entry barriers associated with Grid applications development. GTPE [25] is implemented in pure Java and consists of a thread library that interacts with the Gridbus broker to provide transparent access to Grid services. GTPE provides a finer level of application control as compared to working with coarse grained jobs and frees the developer from the complexities introduced by Grid resource management. Figure 10 illustrates an architecture block-diagram of GTPE. GTPE consists of two main components, the *GridApplication* class and the *GridThread* class.

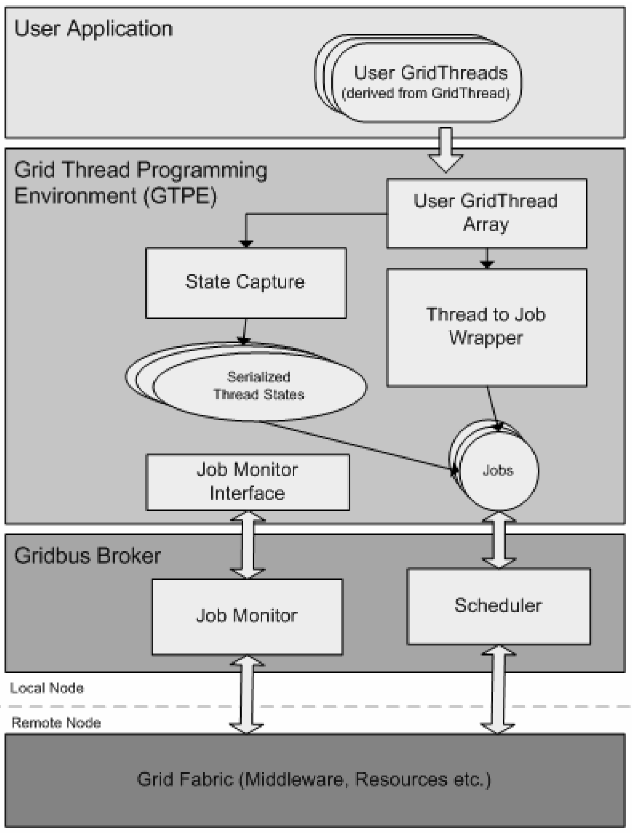


Figure 10: Grid Thread Programming Environment (GTPE) Architecture.

The *GridThread* object forms the atomic unit of remote, independent work. All user defined Grid threads derive from the *GridThread* abstract base class. The subclass has to override the *start* and the *callback* methods. The *start* method is executed on the remote nodes and hence, the computational work intended for remote execution should be defined in this method. The *callback* method is executed at the local client node once the thread has finished executing on the remote node and has been transported back. The *callback* method can be used for a variety of functions including the aggregation of results and the reporting of thread completion to the user.

The *GridApplication* object is responsible for thread management and providing near-transparent access to the Grid via the Gridbus broker. The class presents a single point of control to the programmer. *GridThreads* are added to a *GridApplication* object for execution on remote nodes. The GridApplication object provides mechanisms to capture and restore thread states, as well as job wrapping and thread monitoring services.

The Gridbus-Thread Programming Environment architecture is relatively simple and supports the aforementioned design objectives. GTPE is implemented in pure Java and as such, benefits from its "write-once-run-anywhere" model. User derived Grid threads are inherently portable and able to run on any system which provides access to a Java Virtual Machine. Performance is supported via dynamic scheduling and modern Java compilers, which are able to able to generate Java code capable of execution speeds comparable to traditional high-performance languages [27] . Secure access and job submission to remote nodes is supported via the Gridbus broker and thread monitoring provides a mechanism for detecting thread failures on remote nodes.

8.2.1 Resource Discovery and Access

Version 2.0 of the Gridbus broker (and hence, GTPE) supports the following a range of middleware for computational resources (Globus v2.4 and v3.2, Alchemi v0.8, and Unicore Gateway v41) and data resources (SRB v3.x and Globus Replica Catalog) [3]. If no resources are specified, a set of servers is loaded from a resource file (resources.xml), which specifies default resources and their attributes. The Gridbus broker manages access to these systems, providing secure access via proxies and credentials.

8.2.2 Thread Object State Capture and Job Wrapping

To capture objects into a form which can be distributed, GTPE utilizes Java *serialization*. Serialization is automated by the GridApplication class and is transparent to the user application. However, it is possible for application developers to override the default serialization methods in their derived thread classes to specify optimized implementations that best suit their needs. When a thread is added to for execution on the Grid, it is immediately serialized to a state file with a unique filename. This state file is grouped together with the user derived GridThread class file (obtained using Java class inspection), and the GridThread class file. These files are wrapped into a Gridbus broker job along with the appropriate low-level copy and execute commands. The job object is then added to the Gridbus broker for scheduling and submission to a suitable Grid node.

8.2.3 Thread Scheduling

The Gridbus broker utilizes the concept of a *computational economy* [6] [22] and version 2.0 provides five different scheduling types; cost-optimized, time-optimized, cost-and-time-optimized, cost-and-data-optimized and time-and-data-optimized [12]. If no scheduler is specified, GTPE defaults to using the cost-optimized approach.

8.2.4 Thread Execution and Monitoring

At the remote node, the appropriate user defined subclass is detected and the appropriate thread object instantiated via Java *reflection*. The thread is de-serialized to restore the thread's state and the thread's *start* method is invoked. When the *start* method returns, the thread's state is serialized to a file on disk and transported back to the local node. Each GridThread has an associated status variable which stores one of four possible execution states; *notsubmmited*, *running*, *finished* or *failed*. The default status is the *notsubmmited* state and remains unchanged while it is waiting for transport. When the thread has been submitted to the remote node, its status variable is updated to *running*. If a thread successfully completes, its state is updated by de-serializing the finished thread state file and its status is set to *finished*. If a thread fails during execution, an error report is generated and its status is changed to *failed*.

8.2.5 Additional Functionality

GTPE provides additional functionality to minimize the effort necessary to work with Grid threads. A simple barrier function is implemented allowing users to synchronize threads and the *getThreads* method is available for retrieving threads that have been added to the threads array. These methods are especially useful when aggregating results or performing some final analysis which requires all threads to have completed execution. Hence, unlike regular broker jobs, it is possible to work with updated threads after execution on a remote node. Additionally, the *stop* method is provided to terminate the scheduling and distribution of threads.

9 WORKFLOW

A workflow is a collection of tasks that are executed in some pre-defined order to accomplish a goal. A classic example is the assembly line of a car factory. A Grid workflow is then just a workflow which is executed on the Grid. It should be noted that Grid workflows tend to involve long lasting execution tasks with a large data flow [28]. Grid workflow systems are designed to define, manage and execute Grid workflows. Figure 11 [28] gives an overview of the architecture of a Grid workflow system based on a reference model proposed in 1995.

9.1 Kepler

Kepler [14] is a scientific workflow management system which allows scientists to design, execute and deploy workflows using a number of technologies including web and Grid services, Relational Database Management Systems (RDBMS) and local applications implemented in various programming languages. It is built on top of Ptolemy II, a mature software application developed at the University of California at Berkeley. Ptolemy II [29] is a Java-based component assembly framework with a graphical user interface called Vergil along with a set of Application Program Interfaces (APIs) for heterogeneous hierarchical modeling.

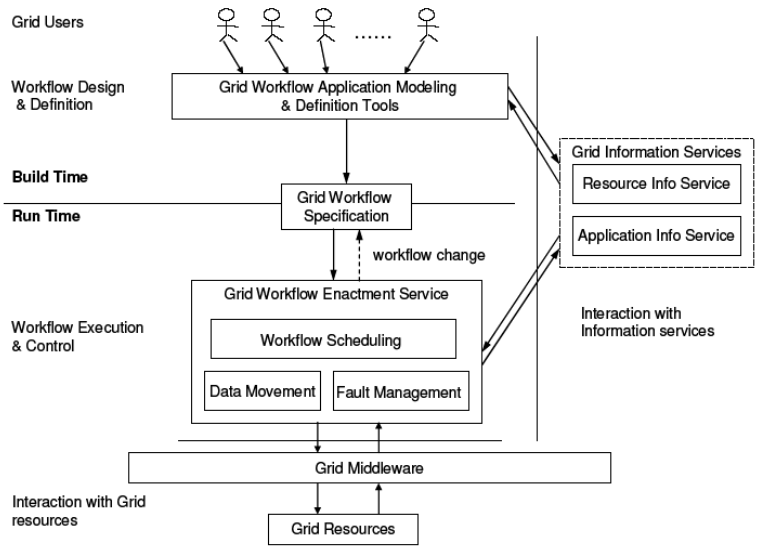

Grid Users
......
Workflow Design & Definition
Grid Workflow Application Modeling & Definition Tools
Build Time
Run Time
Grid Workflow Specification
workflow change
Grid Information Services
Resource Info Service
Application Info Service
Grid Workflow Enactment Service
Workflow Execution & Control
Workflow Scheduling
Data Movement
Fault Management
Interaction with Information services
Grid Middleware
Interaction with Grid resources
Grid Resources

Figure 11: Grid Workflow Management System [28].

9.1.1 Programming aspects of Kepler

Kepler provides domain scientists with an easy to use yet powerful system for capturing scientific workflows via its intuitive programming Graphical User Interface (GUI). It is also a modular, activity oriented programming environment that lends itself to design of reusable components. Its core capabilities that improve the effectiveness and efficiency of scientific research consist of

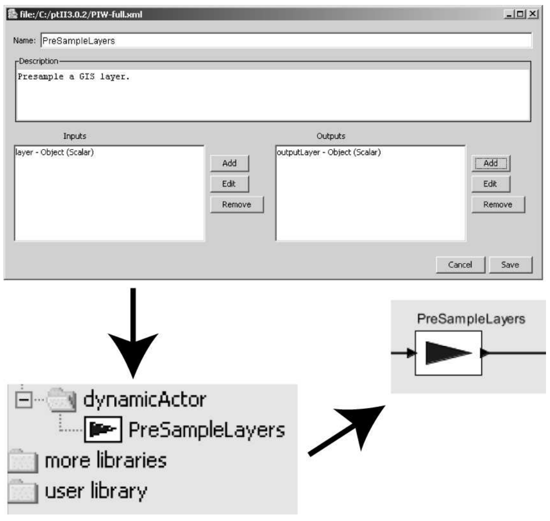
- Capturing Scientific WorkFlow (SWF)s
- Acessing heterogeneous data
- Executing SWFs

9.1.2 Capturing Workflows

In Kepler individual workflow steps are implemented as reusable *actors* [14]. Each actor defines zero or more typed input and output ports that can be linked into a directed graph to allow data flow between actors. Kepler also allows scientists to prototype a workflow before implementing actors needed for the workflow.

Actors: Actors are reusable processing steps that perform computations such as signal processing, statistical operations and Booelan logic operations. Kepler has an extensible library of actors. In-house or third-party software can be added to this library by users. Web and Grid services can be implemented as actors and can be added to the library and used from within Kepler. This is done using the generic Web and Grid Service actors. These actors expose one operation in a Web Service Description Language (WSDL) file or Grid Web Service Description Language (GWSDL) file by exposing the operation's messages as input and output ports. Kepler contains a tool to harvest a group of Web Service descriptions from a repository and save them to the actor library to be used later in workflows. Most actors are Java processes that run locally on a single machine. However, some may call external native applications such as Matlab. Other actors access arbitrary web services that execute a process remotely and return a handle to the results.

Prototyping actors: The actor library may not contain all of the necessary actors to complete a particular scientific computation, so an actor prototyping tool is provided in Kepler (Figure 12 [30]). This tool prompts scientists for critical information about an actor, including its name, icon, and input/output ports. Each port has a name and a data type. Once the user has defined the actor, a stub is compiled and added to the actor library.


file:/C:/ptII3.0.2/PIW-full.xml

Name: PreSampleLayers

Description: Presample a GIS layer.

Inputs: layer - Object (Scalar)

Outputs: outputLayer - Object (Scalar)

Figure 12: The actor prototype tool creates a stub actor class, compiles it, and then adds it to the actor library where it can be dragged onto the workspace [30].

The user can then use this stub on the workflow canvas to prototype a workflow. The ports can be connected to other actors (stubs or not) and the typing system will validate these connections. However, since these actors are stubs that do not implement the intended computations, they simply inform the user that the workflow implementation is incomplete. The stubs must be implemented by writing a Java method for the pre-fire, fire, and post-fire stages of the workflow execution. The intended algorithm can be implemented within the fire event processor or call an external program or service to run. The intent of this tool is to allow a scientist to quickly assemble a workflow without needing to implement the code for every individual piece of the workflow at design time.

Serialization, documentation and provenance: Workflows within Kepler are serialized in an XML dialect called Modeling Markup Language (MoML). XML serialization allows the workflow itself to be used as documentation (metadata) for the research project. The workflow also provides the provenance for derived data products, allowing researchers to return to previous states as needed. The workflow can easily be versioned and archived in any XML storage facility and can be indexed for easy querying and access.

9.1.3 Accessing heterogeneous data

Kepler includes several data access actors including a relational database access actor (*DatabaseQuery*) and a metadata-based data ingestion actor for handling heterogeneous data (*EMLDataSource*). Kepler also provides data transformation facility as actors and web services are generally designed in isolation and therefore input/output incompatibilities are common. Using widely available tools for the two languages, two actors are designed to provide a Kepler interface to XSLT and XQuery. These actors transform XML and HTML data for use in Kepler or outside of Kepler (e.g., browsers).

9.1.4 Executing workflows

Kepler's powerful programming environment supports the varying models of computation that domain scientists may want to use for their models. Kepler can execute processes locally either within the Kepler environment (Java) or within a native environment (compiled native code, or code interpreted by another environment such as Perl). In addition, processes can be executed in a distributed fashion, using web and Grid services. Remotely executed processes behave as a single step in the model of computation regardless of their complexity [30].

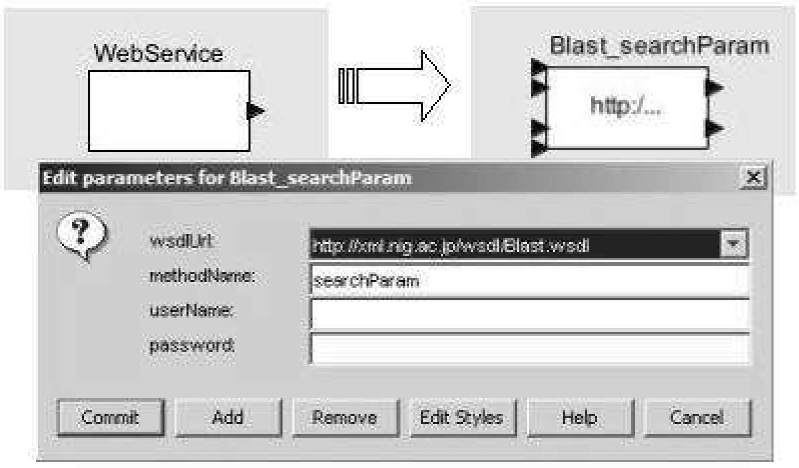
Distributed computation: Kepler's web and Grid services actors allows scientists to utilize computational resources on the network in a distributed scientific workflow. Invocation of each of the distributed services is controlled by the current model of computation in use. The generic *WebService actor* provides the user with an interface to connect to a Web Service defined by a WSDL URL. To customize a Web Service actor, the user provides the URL for the WSDL and selects an operation of the Web Service. The actor automatically customizes its ports with the correct inputs and outputs of the Web Service (Figure 13 [30]), and acts as a proxy to the Web Service when executed. A generic *GridService* actor operates similarly for a given GWSDL URL.

The *Web Service Harvester actor* is used for importing all the operations of a specific Web Service. It can also be used to harvest all of the Web Services in a Universal Description, Discovery and Integration (UDDI) repository. This feature makes it simple for scientists to locate and integrate computational web services of relevance into their computational workflows.

In addition to generic web and Grid services, Kepler includes actors to use Grid based services such as certificate-based authentication (*ProxyInit*), Grid job submission (*GlobusGridJob*), and Grid-based data access (DataAccessWizard). Each of these actors access specific Grid-based services using Open Grid Services Architecture (OGSA) interfaces.

10 GRID SERVICES

In recent years, Web services have gained popularity and importance as a distributed computing paradigm. Its focus is on applying Internet-based standards such as eXtensible Markup Language (XML) to describe the remote software components, the methods by which to access these components and the procedures why which these methods are discovered. These accessible software components are called services and are made available



Edit parameters for Blast_searchParam

wsdlUrl: http://xml.nig.ac.jp/wsdl/Blast.wsdl
methodName: searchParam
userName:
password:

Figure 13: Customizing a Web Service actor [30].

by service providers. In general, a service can be defined as a network enabled entity that provides come capability through the exchange of messages [4]. A *Grid Service* is Web service that provides a set of well defined interfaces and that follows specific conventions [4] [15]. The Grid services paradigm views the Grid as an extensible set of Grid services that can be composed to meet the needs of users, specifically virtual enterprises and virtual organizations [4].

10.1 Open Grid Services Architeture (OGSA)

The Open Grid Services Architecture (OGSA) [4] [15] is an architecture specification defining the semantics and mechanisms governing the creation, access, use, maintenance and destruction of Grid services. In the OGSA world, an entity is represented by a Grid service as defined above. Grid services are *characterized* or *typed* by the capabilities that are offered [4]. Grid services are accessed via standard interface(s) defined by OGSA and as such, these services can be composed to form more sophisticated services. Each interface consists of a set of operation definitions that are invoked by a defined sequence of exchanged messages. Note that OGSA does not address how these services are actually implemented i.e. what the service is or how it performs its function(s). This allows for a flexible approach to service construction, as long as the service complies with the semantics outlined in the OGSA specification.

10.1.1 Grid Service Instances

Grid services maintain an internal state for the lifetime of the service and it via this internal state that *instances* of services are differentiated from one another. Typically, Grid service users would not require a static set of persistent services but would rather need to instantiate service instances dynamically. As such, OGSA provides specifications for the management of *transient service instances*, i.e. temporary service instances which can be destroyed after being used. OGSA specifies the *Factory* interface for the purpose of creating a requested Grid service. Each Grid service instance is assigned a global Grid service handle (GSH), by which it is uniquely identified.

10.1.2 Upgradeability and Communication

The GSH provides only a unique identifier for each instance and does not include protocol or instance specific information required to interact with the instance. This information is contained within another entity termed the Grid service reference (GSR). Separating these two information entities allows for the instance to be changed or upgraded over its lifetime without requiring a new unique identifier. While an instance's GSH is static, its GSR may change. To obtain a valid GSR from a GSH, OGSA defines a handle-to-reference mapper interface (*HandleMap*).

10.1.3 Service Discovery

GSHs can be initially obtained via a *registry*, which is a grid service that supports service discovery. A registry service is defined by the *Registry* interface and a service data element containing information relating to registered GSHs. The *Registry* interface provides a set of operations that allow the registration of a GSH. Information regarding registered GSHs is obtained via the *GridService* interface's FindServiceData operation.

10.1.4 Notification

Services are able to provide notification messages to clients by supporting the *NotificationSource* interface to manage notification request subscriptions. A client can invoke this service using the subscribe operation and providing the GSH of the notification destination (termed the notification sink). The sink is required to send the notification source periodic keepalive messages to continue to receive notifications. Keepalive messages are also used to manage the lifetime of a transient service.

10.1.5 Service Lifetime Management

The introduction of transient services poses the problem of service lifetime determination and management. As the grid is open and dynamic, various components may fail and a created service may not be explicitly terminated by the client. OGSA solves this problem via a soft state approach [31] which consist of operations for the negotiation of an initial lifetime, extension requests and service instance harvest after lifetime expiry. During the initial lifetime negotiation, the client specifies a minimum and maximum acceptable initial lifetimes. From this range, the factory selects an initial lifetime which is returned to the client. A client can extend a lifetime by specifying a new minimum and maximum lifetime using the SetTerminationTime message i.e. a keepalive message. If a service instance lifetime expires, the host environment terminates the service and reclaims the associated resources.

10.1.6 Higher Level Capabilities

The latest specification of OGSA [15] at the time of writing also specifies a wide range of higher level capabilities including Execution Management Services (EMS), Data Services, Resource Management Services, Security Services, Self-Management Services and Information Services. We refer readers requiring more detail regarding these services to [15].

11 SUMMARY AND CONCLUSION

In this chapter we have considered various programming models and environments for the development of Grid applications. We began by motivating the need for Grid programming environments that make transparent the heterogeneous and dynamic nature of the Grid. A successful Grid programming environment should support portability, interoperability, adaptivity, discovery, security and fault tolerance while maintaining high performance.

We discussed an implicit parallelization environment, Grid superscalar, two explicit parallelization environments, MPICH-G2 and Ninf-G GridRPC and finally semi-parallelization environments, the Gridbus Broker, Alchemi, GTPE and the Kepler workflow system. It is tempting to compare these systems directly in an attempt to deduce the best system for general purpose use. However, we believe that such a comparison would be premature. All the systems we reviewed yield promising results in terms of usability and performance. The choice of a Grid programming model depends largely on the task at hand and the skills available to the application developers. An application developer who is familiar with the MPI method of programming clusters may find the transition to a Grid-enabled MPI implementation to be simpler than implementing Grid applications using an automated but unfamiliar system.

What we believe to be important is not which model will emerge as the dominant system but that there are already exist tools available for us to develop applications that are able to take advantage of the incredible computational power of the Grid. The diversity of the models provides us with choice and assures us that there is likely a model that best suits a particular application. In addition, all of the programming environments discussed are works in progress and we expect advances in all areas yielding better systems over time. This is an exciting time in this area of research and while much work needs to be done, the results thus far are very promising.

ACKNOWLEDGEMENT

We would like to thank Marco A. S. Netto for typesetting this chapter using Latex.

REFERENCES

[1] I. Foster and C. Kesselman, *The grid: blueprint for a new computing infrastructure*: Morgan Kaufmann Publishers Inc., 1999.

[2] P. Asadzadeh, R. Buyya, C. L. Kei, D. Nayar, and S. Venugopal, “Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies”, High Performance Computing: Paradigm and Infrastructure, Laurence Yang and Minyi Guo (eds), pp.431-458 (Chapter 22), ISBN: 0-471-65471-X, Wiley Press, New Jersey, USA, June 2005.

[3] I. Foster, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations”, *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*: Springer-Verlag, 2001, pp. 1-4.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration”, 2002. URL: www.globus.org/research/papers/ogsa.pdf

[5] I. Foster and C. Kesselman, “The Globus toolkit”, *The grid: blueprint for a new computing infrastructure* Morgan Kaufmann Publishers Inc., 1999 pp. 259-278

[6] R. Buyya, D. Abramson, and J. Giddy, “An Economy Driven Resource Management Architecture for Global Computational Power Grids”, Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, USA, 2000

[7] R. Buyya, D. Abramson, and J. Giddy, “Nimrod/G: an architecture for a resource management and scheduling system in a global computational grid”, Proceedings of the 4th International Conference on High Performance Computing in the Asia-Pacific Region, May 2000.

[8] R. M. Badia, J. S. Labarta, R. L. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima, “Programming Grid Applications with GRID Superscalar”, *Journal of Grid Computing*, vol. 1, pp. 151-170, 2003.

[9] N. T. Karonis, B. Toonen, and I. Foster, “MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface”, *Journal of Parallel and Distrbuted Computing (JPDC)*, vol. 63, pp. 551-563, 2002.

[10] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, “Overview of GridRPC: A Remote Procedure Call API for Grid Computing”, Proceedings of the Third International Workshop on Grid Computing, Lecture Notes in Computer Science, Springer, ed 2536, pp. 274–278, Baltimore, USA, November 2002.

[11] S. Venugopal, R. Buyya, and L. Winton, “A grid service broker for scheduling distributed data-oriented applications on global grids”, Proceedings of the 2nd workshop on Middleware for grid computing, pp. 75–80, Toronto, Canada, 2004.

[12] K. Nadiminti, S. Venugopal, H. Gibbins, T. Ma, and R. Buyya, “The Gridbus Grid Service Broker and Scheduler (v.2.2) User Guide”, Grid Computing and Distributed Systems (GRIDS) Laboratory, Department of Computer Science and Software Engineering, The University of Melbourne, Australia 2005.

[13] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal, “Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework”, *High Performance Computing: Paradigm and Infrastructure*, L. Y. a. M. Guo, Ed.: Wiley Press, 2005.

[14] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, “Kepler: an extensible system for design and execution of scientific workflows”, Proceedings of the 16th International Conference on Scientific and Statistical Database Management(SSDBM), Santorini Island, Greece, 2004

[15] J. Frey, T. Mori, J. Nick, C. Smith, D. Snelling, L. Srinivasan, and J. Unger, “The Open Grid Services Architecture, Version 1.0”, 2005. URL: https://forge.gridforum.org/projects/ogsa-wg

[16] “CORBA 3.0 - OMG IDL Syntax and Semantics chapter”, Object Management Group 2005. URL: http://www.omg.org/cgi-bin/doc?formal/02-06-39

[17] “MPI: A Message-Passing Interface Standard”, The Message Passing Interface Forum, 1995.

[18] “MPI-2: Extensions to the Message-Passing Interface”, The Message Passing Interface Forum, 1997.

[19] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard”, *Parallel Computing*, vol. 22, pp. 789-828, 1996

[20] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova, “A GridRPC Model and API”, Global Grid Forum, 2003.

[21] Y. T. Hidemoto Nakada, Satoshi Matsuoka, Satoshi Sekiguchi, “Ninf-G: A GridRPC System on the Globus Toolkit”, *Grid Computing*, G. F. Fran Berman, Tony Hey, Ed., 2003, pp. 625-637.

[22] R. Buyya, K. Branson, J. Giddy, and D. Abramson, “The Virtual Laboratory: Enabling Molecular Modeling for Drug Design on the World Wide Grid”, Concurrency and Computation: Practice and Experience (CCPE), Volume 15, Issue 1, Pages: 1-25, Wiley Press, USA, January 2003.

[23] ProactiveTeam, “Proactive Manual REVISED 2.2”, Proactive, INRIA April 2005.

[24] D. T. Craig Lee, “Grid Programming Models: Current Tools, Issues and Directions”, *Grid Computing*, G. F. Fran Berman, Tony Hey, Ed., pp. 555–578, Wiley Press, USA, 2003.

[25] H. Soh, S. Haque, W. Liao, K. Nadiminti, and R. Buyya, “GTPE: A Thread Programming Environment for the Grid”, Proceedings of the 13th International Conference on Advanced Computing and Communications, Coimbatore, India, 2005

[26] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal, “Alchemi: A .NET-Based Enterprise Grid Computing System”, Proceedings of the 6th International Conference on Internet Computing (ICOMP’05), June 27-30, 2005, Las Vegas, USA.

[27] J. Bull, L. Smith, P. L, and R. Freeman, “Benchmarking Java against C and Fortran for Scientific Applications”, Proceedings of the ACM 2001 Java Grande/ISCOPE Conference, 2001.

[28] J. Yu and R. Buyya, “A Taxonomy of Workflow Management Systems for Grid Computing”, Journal of Grid Computing, volume 3, numbers 3-4, pp. 171-200, Springer Science+Business Media B.V., New York, USA, September 2005.

[29] E. A. Lee, “Overview of the Ptolemy Project”, Center for Hybrid and Embedded Software Systems (CHESS), University of California, Berkeley. UCB/ERL M03/25, 2003.

[30] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, “Kepler: Towards a Grid-Enabled System for Scientific Workflows”, Workflow in Grid Systems Workshop in GGF10 - The Tenth Global Grid Forum, Berlin, Germany, 2004

[31] D. D. Clark, “The Design Philosophy of the DARPA Internet Protocols”, *SIGCOMM Symposium on Communications Architectures and Protocols*, pp. 106-114, 1988.