Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework

Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal

Grid Computing and **D**istributed **S**ystems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
Email:{akshayl, raj, rranjan, srikumar}@cs.mu.oz.au

Abstract: Computational grids that couple geographically distributed resources are becoming the de-facto computing platform for solving large-scale problems in science, engineering, and commerce. Software to enable grid computing has been primarily written for Unix-class operating systems, thus severely limiting the ability to effectively utilize the computing resources of the vast majority of Windows-based desktop computers. Addressing Windows-based enterprise grid computing is particularly important from the software industry's viewpoint where interest in grids is emerging rapidly. Microsoft's .NET Framework has become near-ubiquitous for implementing commercial distributed systems for Windows-based platforms, positioning it as the ideal platform for developing peer-to-peer or enterprise grid computing environments. This chapter introduces design requirements of enterprise grid systems and discusses various middleware technologies that meet them. It presents a .NET-based Grid framework, called Alchemi developed as part of the Gridbus project. Alchemi provides the runtime machinery and programming environment required to construct enterprise grids and develop grid applications. It allows flexible application composition by supporting an object-oriented application programming model in addition to a file-based job model. Cross-platform support is provided via a web services interface and a flexible execution model supports dedicated and non-dedicated execution by grid nodes.

Keywords: Peer-to-peer computing, enterprise computing, grid computing, Web services, .NET, and grid application programming.

1 Introduction

The idea of metacomputing [2] is very promising as it enables the use of a network of many independent computers as if they were one large parallel machine, or virtual supercomputer at a fraction of the cost of traditional supercomputers. While traditional virtual machines (e.g. clusters) have been designed for local area networks, the exponential growth in Internet connectivity allows this concept to be applied on a much larger scale. This, coupled with the fact that desktop PCs (personal computers) in corporate and home environments are heavily underutilized – typically only one-tenth of processing power is used – has given rise to interest in harnessing these unused CPU cycles of desktop PCs connected over the Internet [20]. This new paradigm has been dubbed as peer-to-peer (P2P) computing [18], which is being recently called enterprise desktop grid computing [17].

Although the notion of desktop grid computing is simple enough, the practical realization of a peer-to-peer grid poses a number of challenges. Some of the key issues include: heterogeneity, resource management, failure management, reliability, application composition, scheduling and security [13]. Further, for wide-scale adoption, desktop grid computing infrastructure must also leverage the power of Windows-class machines since the vast majority of desktop computers run variants of the Windows operating system.

However, there is a distinct lack of service-oriented architecture-based grid computing software in this space. To overcome this limitation, we have developed a Windows-based desktop grid computing framework called Alchemi implemented on the Microsoft .NET Platform. The Microsoft .NET Framework is the state of the art development platform for Windows and offers a number of features which can be leveraged for enabling a computational desktop grid environment on Windows-class machines.

Alchemi was conceived with the aim of making grid construction and development of grid software as easy as possible without sacrificing flexibility, scalability, reliability and extensibility. The key features supported by Alchemi are:

- Internet-based clustering [21][22] of Windows-based desktop computers;
- dedicated or non-dedicated (voluntary) execution by individual nodes;
- object-oriented grid application programming model (fine-grained abstraction);
- file-based grid job model (coarse-grained abstraction) for grid-enabling legacy applications and
- web services interface supporting the job model for interoperability with custom grid middleware e.g. for creating a global, cross-platform grid environment via a custom resource broker component.

The rest of the chapter is organized as follows. Section 2 presents background information on P2P and grid computing and Section 3 discusses a basic architecture of enterprise desktop Grid system along with middleware design considerations. Section 4 introduces desktop grids and discusses issues that must be addressed by a desktop grid. Section 4 briefly presents various enterprise grid systems along with their comparison to our Alchemi middleware. Section 5 presents the Alchemi desktop grid computing framework and describes its architecture, application composition models and its features with respect to the requirements of a desktop grid solution. Section 6 deals with the system implementation and presents the lifecycle of an Alchemi-enabled grid application demonstrating its execution model. Section 6 presents the results of an evaluation of Alchemi as a platform for execution of applications written using the Alchemi API. It also evaluates the use of Alchemi nodes as part of a global grid alongside Unix-class grid nodes running Globus software. Finally, we conclude the chapter with work planned for the future.

2 Background

In the early 1970s when computers were first linked by networks, the idea of harnessing unused CPU cycles was born [34]. A few early experiments with distributed computing—including a pair of programs called *Creeper and Reaper*—ran on the Internet's predecessor, the ARPAnet. In 1973, the Xerox Palo Alto Research Center (PARC) installed the first Ethernet network and the first fully-fledged distributed computing effort was underway. Scientists at PARC developed a program called "worm" that routinely cruised about 100 Ethernet-connected computers. They envisioned their worm migrating from machine to another harnesses idle resources for beneficial purposes. The worm would roam throughout the PARC network, replicating itself in each machine's memory. Each worm used idle resources to perform a computation and had the ability to reproduce and transmit clones to other nodes of the network. With the worms, developers distributed graphic images and shared computations for rendering realistic computer graphics.

Since 1990, with the maturation and ubiquity of the Internet and Web technologies along with availability of powerful computers and system area networks as commodity components, distributed computing scaled to a new global level. The availability of powerful PCs and workstations; and high-speed networks (e.g., Gigabit Ethernet) as commodity components has lead to the emergence of clusters [35] serving the needs of high performance computing (HPC) users. The ubiquity of the Internet and Web technologies along with the availability of many low-cost and high-performance commodity clusters within many organizations has prompted the exploration of aggregating distributed resources for solving large scale problems of multi-institutional interest. This has led to the emergence of computational Grids and P2P networks for sharing distributed resources. The grid community is generally focused on aggregation of distributed high-end machines such as clusters whereas P2P community is looking into sharing low-end systems such as PCs connected to the Internet for sharing computing power (e.g., SETI@Home) and contents (e.g., exchange music files via Napster and Gnuetella networks). Given the number of projects and forums [36][37] started all over the world in early 2000, it is clear that the interest in the research, development, and deployment of Grid and P2P computing technologies, tools, and applications is rapidly growing.

3 Desktop Grid Middleware Considerations

Figure 1 shows the architecture of a basic desktop grid computing system. Typically, users utilize the API's and tools to interact with a particular grid middleware to develop grid applications. When they submit grid application for processing, units of work are submitted to a central controller component which co-

ordinates and manages the execution of these work units on the worker nodes under its control. There are a number of considerations that must be addressed for such a system to work effectively.

Security Barrier - Resource Connectivity behind Firewalls
Firstly, worker nodes and user nodes must be able to connect to the central controller over the Internet or LAN and the presence of firewalls and/or NAT servers must not affect the deployment of a desktop grid.

Unobtrusiveness - No Impact on Running User Applications
The execution of grid applications by worker nodes must not affect running user programs.

Programmability - Computationally Intensive Independent Work Units
As desktop grid systems span across high latency of the Internet environment, applications with a high ratio of computation to communication time are suitable for deployment and are thus typically embarrassingly parallel.

Reliability – Failure Management
The unreliable nature of Internet connections also means that such systems must be able to tolerate connectivity disruption or faults and recover from them gracefully. In addition, data loss must be minimized in the event of a system crash or failure.

Scalability – Handle Large Users and Participants
Desktop grid systems must be designed to support the participation of a large number of anonymous or approved contributors ranging from hundreds to millions. In addition, the system must support a number of simultaneous users and their applications.

Security – Protect both Contributors and Consumers
Finally, the Internet is an insecure environment and strict security measures are imperative. Specifically, users and their programs must only be able to perform authorized activities on the grid resources. In addition, users/consumers must be safeguarded against malicious attacks or worker nodes.

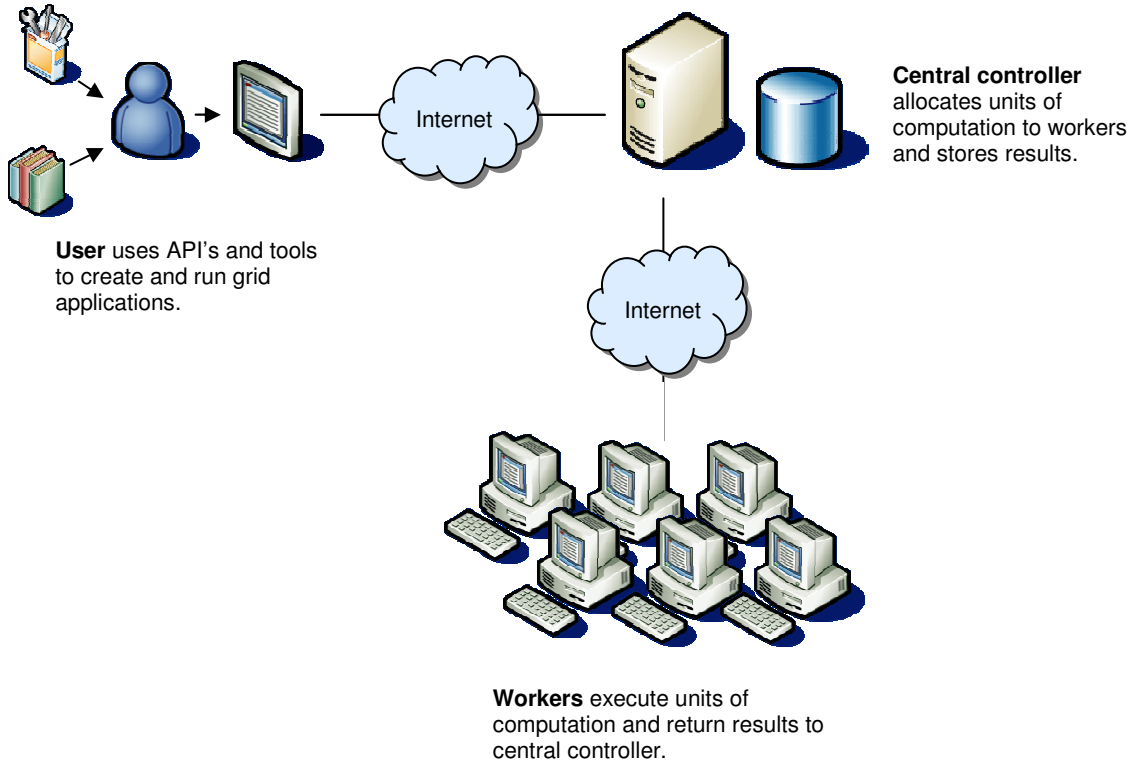

Figure 1. Architecture of a basic desktop grid.

4 Representative Desktop Grid Systems

In addition to its implementation based on a service-oriented architecture using state-of-the-art technologies, Alchemi has a number of distinguished features when compared to related systems. Table 2 shows a comparison between Alchemi and some related systems such as Condor, SETI@home, Entropia, GridMP, and XtermWeb.

Alchemi is a .NET-based framework that provides the runtime machinery and programming environment required to construct desktop grids and develop grid applications. It allows flexible application composition by supporting an object-oriented application programming model in addition to a file-based job model. Cross-platform support is provided via a web services interface and a flexible execution model supports dedicated and non-dedicated (voluntary) execution by grid nodes.

Condor [19] system is developed by the University of Wisconsin at Madison. It can be used to manage a cluster of dedicated or non-dedicated compute nodes. In addition, unique mechanisms enable Condor to effectively harness wasted CPU power from otherwise idle desktop workstations. Condor provides a job queuing mechanism, scheduling policy, workflow scheduler, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion. It can handle both Windows and UNIX class resources in its resource pool. Recently Condor has been extended (see Condor-G [38]) to support the inclusion of Grid resources within a Condor pool.

System / Property	Alchemi	Condor	SETI@home	Entropia	XtermWeb	Grid MP
Architecture	Hierarchical	Hierarchical	Centralized	Centralized	Centralized	Centralized
Web Services Interface for Cross-Platform Integration	Yes	No	No	No	No	Yes
Implementation Technologies	C#, Web Services, & .NET Framework	C	C++, Win32	C++, Win32	Java, Linux	C++, Win32
Multi-Clustering	Yes	Yes	No	No	No	Yes
Global Grid Brokering Mechanism	Yes (via Gridbus Broker)	Yes (via Condor-G)	No	No	No	No
Thread Programming Model	Yes	No	No	No	No	No
Level of integration of application, programming and runtime environment	Low (general purpose)	Low (general purpose)	High (single purpose, single application environment)	Low (general purpose)	Low (general purpose)	Low (general purpose)

Table 2. Comparison of Alchemi and some related desktop grid systems.

The Search for Extraterrestrial Intelligence (SETI) project [9][14], named SETI@Home, based at the University of California at Berkeley is aimed at doing good science in such a way that it engages and excites the general public. It developed a desktop grid system that harnesses hundreds and thousands of PCs across the Internet to processing a massive amount of astronomy data captured daily by the Arecibo telescope in Puerto Rico. Its worker software runs as a screen saver on contributor computers. It is designed to work on heterogeneous computers running Windows, Mac, and variants of UNIX operating systems. Unlike other desktop systems, the worker module is designed as application specific software as it supports processing of astronomy application data only.

Entropia [17] facilitates a Windows desktop grid system by aggregating the raw desktop resources into a single logical resource. Its core architecture is centralized in which a central job manager administers various desktop clients. The node manager provides a centralized interface to manage all of the clients on the Entropia grid, which is accessible from anywhere on the enterprise network.

XtermWeb [16] is a P2P [15][18] system developed at the University of Paris-Sud, France. It implements three distinct entities, the coordinator, the workers and the clients to create a so-called XtermWeb network. Clients are software instances available for any user to submit tasks to the XtermWeb network. They submit tasks to the coordinator, providing binaries and optional parameter files and permit the end user to retrieve results. Finally, the workers are software parts spread among volunteer hosts to compute tasks.

The Grid MP (MP) [23] is developed by United Devices whose expertise is mainly drawn through the recruitment of key developers of SETI@Home and Distributed.Net enterprise grid system. Like other systems, it supports harnessing and aggregation compute resources available on their corporate network. It basically has a centralized architecture, where a Grid MP service acting as a manager accepts jobs from the user, schedules them on the resources having pre-deployed Grid MP agents. The Grid MP agents can be deployed on clusters, workstations or desktop computers. Grid MP agents receive jobs and execute them on resources, advertise their resource capabilities on Grid MP services and return results to the Grid MP services for subsequent collection by the user.

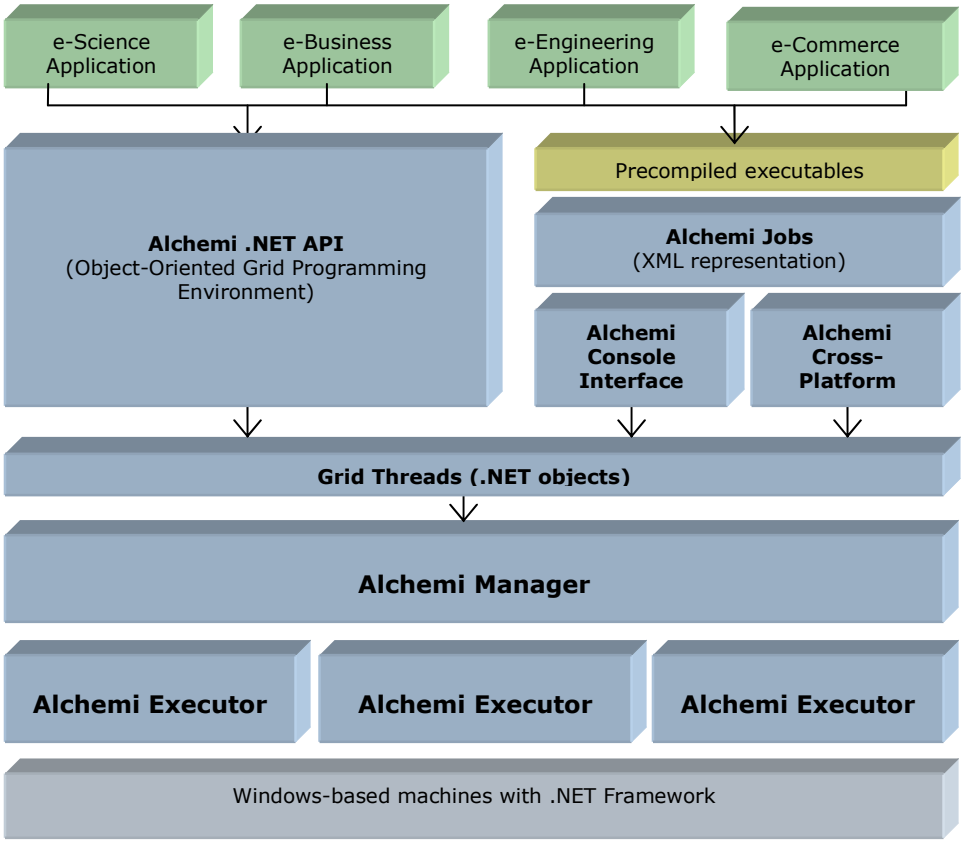

Figure 2. A layered architecture for a desktop grid computing environment.

5 Alchemi Desktop Grid Framework

Alchemi's layered architecture for a desktop grid computing environment is shown in Figure 2. Alchemi follows the master-worker parallel computing paradigm [31] in which a central component dispatches

independent units of parallel execution to workers and manages them. In Alchemi, this unit of parallel execution is termed 'grid thread' and contains the instructions to be executed on a grid node, while the central component is termed 'Manager'.

A 'grid application' consists of a number of related grid threads. Grid applications and grid threads are exposed to the application developer as .NET classes / objects via the Alchemi .NET API. When an application written using this API is executed, grid thread objects are submitted to the Alchemi Manager for execution by the grid. Alternatively, file-based jobs (with related jobs comprising a task) can be created using an XML representation to grid-enable legacy applications for which precompiled executables exist. Jobs can be submitted via Alchemi Console Interface or Cross-Platform Manager web service interface, which in turn convert them into the grid threads before submitting then to the Manager for execution by the grid.

5.1 Application Models

Alchemi supports functional and well as data parallelism. Both are supported by each of the two models for parallel application composition – grid thread model and grid job model.

5.1.1 Grid Thread Model

Minimizing the entry barrier to writing applications for a grid environment is one of Alchemi's key goals. This goal is served by an object-oriented programming environment via the Alchemi .NET API which can be used to write grid applications in any .NET-supported language.

The atomic unit of independent parallel execution is a grid thread with many grid threads comprising a grid application (hereafter, 'applications' and 'threads' can be taken to mean grid applications and grid threads respectively, unless stated otherwise). The two central classes in the Alchemi .NET API are `GThread` and `GApplication`, representing a grid thread and grid application respectively. There are essentially two parts to an Alchemi grid application. Each is centered on one of these classes:

- "Remote code": code to be executed remotely i.e. on the grid (a grid thread and its dependencies) and
- "Local code": code to be executed locally (code responsible for creating and executing grid threads).

A concrete grid thread is implemented by writing a class that derives from `GThread`, overriding the `void Start()` method, and marking the class with the `Serializable` attribute. Code to be executed remotely is defined in the implementation of the overridden `void Start()` method.

The application itself (local code) creates instances of the custom grid thread, executes them on the grid and consumes each thread's results. It makes use of an instance of the `GApplication` class which represents a grid application. The modules (.EXE or .DLL files) containing the implementation of this `GThread`-derived class and any other dependency types that not part of the .NET Framework must be included in the `Manifest` of the `GApplication` instance. Instances of the `GThread`-derived class are asynchronously executed on the grid by adding them to the grid application. Upon completion of each thread, a 'thread finish' event is fired and a method subscribing to this event can consume the thread's results. Other events such as 'application finish' and 'thread failed' can also be subscribed to. Thus, the programmatic abstraction of the grid in this manner described allows the application developer to concentrate on the application itself without worrying about "plumbing" details.

Appendix A shows the entire code listing of a sample application for multiplying pairs of integers.

5.1.2 Grid Job Model

Traditional grid implementations have offered a high-level, abstraction of the "virtual machine", where the smallest unit of parallel execution is a process. In this model, a work unit is typically described by specifying a command, input files and output files. In Alchemi, such a work unit is termed 'job' with many jobs constituting a 'task'.

Although writing software for the "grid job" model involves dealing with files, an approach that can be complicated and inflexible, Alchemi's architecture supports it for the following reasons:

- grid-enabling existing applications; and
- interoperability with grid middleware that can leverage Alchemi via the Cross Platform Manager web service

Tasks and their constituent jobs are represented as XML files conforming to the Alchemi task and job schemas. Figure 3 shows a sample task representation that contains two jobs to execute the **`Reverse.exe`** program against two input files.

```
<task>
  <manifest>
    <embedded_file name="Reverse.exe" location="Reverse.exe" />
  </manifest>

  <job id="0">
    <input>
      <embedded_file name="input1.txt" location="input1.txt" />
    </input>
    <work run_command="Reverse.exe input1.txt > result1.txt" />
    <output>
      <embedded_file name="result1.txt"/>
    </output>
  </job>

  <job id="1">
    <input>
      <embedded_file name="input2.txt" location="input2.txt" />
    </input>
    <work run_command="Reverse input2.txt > result2.txt" />
    <output>
      <embedded_file name="result2.txt"/>
    </output>
  </job>
</task>
```

Figure 3. Sample XML-based task representation.

Before submitting the task to the Manager, references to the 'embedded' files are resolved and the files themselves are embedded into the task XML file as Base64-encoded text data. When finished jobs are retrieved from the Manager, the Base64-encoded contents of the 'embedded' files are decoded and written to disk.

It should be noted that tasks and jobs are represented internally as grid applications and grid threads respectively. Thus, any discussion that applies to 'grid applications' and 'grid threads' applies to 'grid tasks' and 'grid jobs' as well.

5.2 Distributed Components

Four types of nodes (or hosts) take part in desktop grid construction and application execution (see Figure 4). An Alchemi desktop grid is constructed by deploying a Manager node and deploying one or more Executor nodes configured to connect to the Manager. One or more Users can execute their applications on the cluster by connecting to the Manager. An optional component, the Cross Platform Manager provides a web service interface to custom grid middleware. The operation of the Manager, Executor, User and Cross Platform Manager nodes is described below.

5.2.1 Manager

The Manager provides services associated with managing execution of grid applications and their constituent threads. Executors register themselves with the Manager, which in turn monitors their status. Threads received from the User are placed in a pool and scheduled to be executed on the various available

Executors. A priority for each thread can be explicitly specified when it is created or submitted. Threads are scheduled on a Priority and First Come First Served (FCFS) basis, in that order. The Executors return completed threads to the Manager which are subsequently collected by the respective users. A scheduling API is provided that allows custom schedulers to be written.

The Manager employs a role-based security model for authentication and authorization of secure activities. A list of permissions representing activities that need to be secured is maintained within the Manager. A list of groups (roles) is also maintained, each containing a set of permissions. For any activity that needs to be authorized, the user or program must supply credentials in a form of a user name and password and the Manager only authorizes the activity if the user belongs to a group that contains the particular permission.

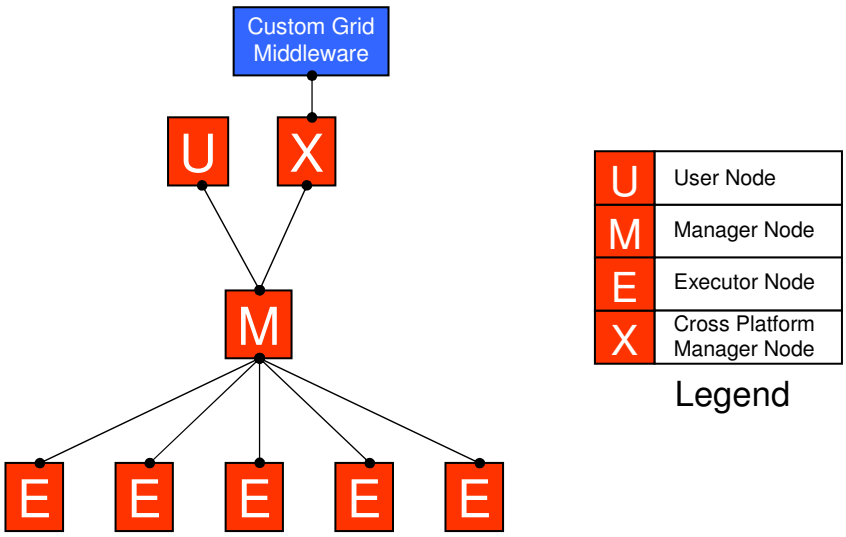

Figure 4. Distributed components and their relationships.

As discussed previously, failure management plays a key role in the effectiveness of a desktop grid. Executors are constantly monitored and threads running on disconnected Executors are rescheduled. Additionally, all data is immediately persisted to disk so that in the event of a crash, the Manager can be restarted into the pre-crash state.

5.2.2 Executor

The Executor accepts threads from the Manager and executes them. An Executor can be configured to be dedicated, meaning the resource is centrally managed by the Manager, or non-dedicated, meaning that the resource is managed on a volunteer basis via a screen saver or explicitly by the user. For non-dedicated execution, there is one-way communication between the Executor and the Manager. In this case, the resource that the Executor resides on is managed on a volunteer basis since it requests threads to execute from the Manager. When two-way communication is possible and dedicated execution is desired the Executor exposes an interface so that the Manager may communicate with it directly. In this case, the Manager explicitly instructs the Executor to execute threads, resulting in centralized management of the resource where the Executor resides. Thus, Alchemi's execution model provides the dual benefit of:

- flexible resource management i.e. centralized management with dedicated execution vs. decentralized management with non-dedicated execution; and
- flexible deployment under network constraints i.e. the component can be deployment as non-dedicated where two-way communication is not desired or not possible (e.g. when it is behind a firewall or NAT/proxy server).

Thus, dedicated execution is more suitable where the Manager and Executor are on the same Local Area Network while non-dedicated execution is more appropriate when the Manager and Executor are to be connected over the Internet.

Threads are executed in a sandbox environment defined by the user. The CAS (Code Access Security) feature of .NET are used to execute all threads with the *AlchemiGridThread* permission set which can be specified to a fine-grained level by the user as part of the .NET *Local Security Policy*.

All grid threads run in the background with the lowest priority. Thus any user programs are unaffected since they have higher priority access to the CPU over grid threads.

5.2.3 User

Grid applications are executed on the User node. The API abstracts the implementation of the grid from the user and is responsible for performing a variety of services on the user's behalf such as submitting an application and its constituent threads for execution, notifying the user of finished threads and providing results and notifying the user of failed threads along with error details.

5.2.4 Cross-Platform Manager

The Cross-Platform Manager is a web services interface that exposes a portion of the functionality of the Manager in order to enable Alchemi to manage the execution of grid jobs (as opposed to grid applications utilizing the Alchemi grid thread model). Jobs submitted to the Cross-Platform Manager are translated into a form that is accepted by the Manager (i.e. grid threads), which are then scheduled and executed as normal in the fashion described above. In addition to support for the grid-enabling of legacy applications, the Cross-Platform Manager allows custom grid middleware to interoperate with and leverage Alchemi on any platform that supports web services.

5.3 Security

Security plays a key role in an insecure environment such as the Internet. Two aspects of security addressed by Alchemi are: (a) allow users to perform authorized operations whether they are system related or resource related operations and (b) allow authorized or non-authorized users to contribute resources.

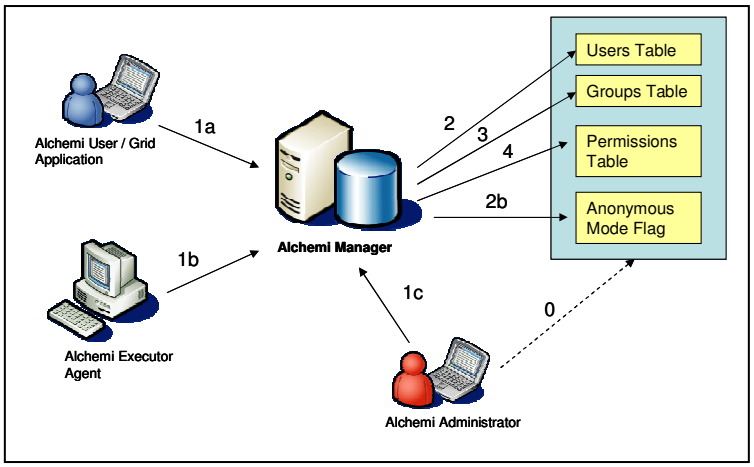

Figure 5. Role-based security in Alchemi.

The problem of allowing users to only perform activities they are authorized to do is addressed using the role-based authorization model. All security-sensitive activities on the Manager are protected in this manner. The Manager can be configured to support anonymous or non-anonymous Executors. Figure 5 shows the operation of various Alchemi components to enforce security as indicated below.

0: The Alchemi administrator configures user, group and permission data in addition to allowing anonymous/non-anonymous Executors.
1: A user or program connects to the Manager, supplies security credentials and requests a particular activity.
2, 3, 4: The Manager authenticates the user and authorizes the user for the activity. This process is skipped if anonymous Executors are allowed.

Information about permission, groups and users is maintained in the Alchemi database in the *prm*, *grp* and *usr* tables respectively. Figure 6 shows the relationships between these tables.
Each activity on the Manager is associated with a particular permission. The following permissions are defined:

- *ExecuteThread* (activities related to thread execution, e.g. getting a thread to execute, returning its results)
- *ManageOwnApp* (activities related to the ownership of a particular application, e.g. creating an application, getting its finished threads)
- *ManageAllApps* (activities related to the ownership of all applications in the system e.g. getting a list of all applications along with statistics)
- *ManageUsers* (activites related to user management, e.g. adding users, changing passwords, changing group membership)

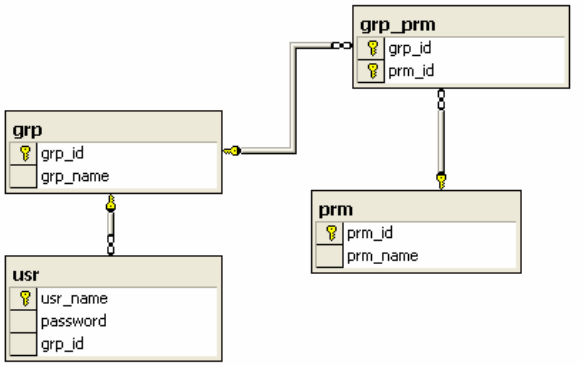

Figure 6. Relationships between security-related database tables.

Users belong to a particular group, with each group containing a set of permissions. The following groups are defined:

- *Users* (*ManageOwnApp*)
- *Executors* (*ExecuteThread*)
- *Administrators* (*ManageOwnApp*, *ManageAllApps*, *ExecuteThread*, *ManageUsers*)

For any activity that needs to be authorized, the user or program must supply credentials in a form of a user name and password and the Manager only authorizes the activity if the user belongs to a group that contains the particular permission. Figure 7 shows the process of authentication and authorization.

The second aspect of security that Alchemi addresses is the protection of the machine hosting the Executor from malicious code. This is solved by the creation of a "sandbox" environment in which the Executor runs grid threads. This environment can be defined by the user. The CAS (Code Access Security) feature of .NET is used to execute all threads with the *AlchemiGridThread* permission set which can be specified to a fine-grained level by the user as part of the .NET *Local Security Policy*.

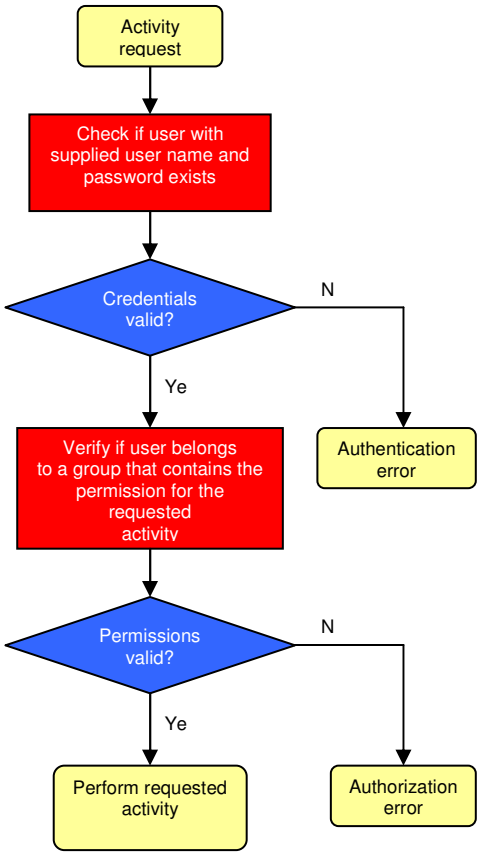

Figure 7. Authorization flow.

6 Alchemi Design and Implementation

Figure 8 and Figure 9 provide an overview of the implementation by way of a deployment diagram and class diagram (showing only the main classes without attributes or operations) respectively.

6.1 Overview

The .NET Framework offers two mechanisms for execution across application domains – Remoting and web services (application domains are the unit of isolation for a .NET application and can reside on different network hosts).

.NET Remoting allows a .NET object to be "remoted" and expose its functionality across application domains. Remoting is used for communication between the four Alchemi distributed grid components as it allows low-level interaction transparently between .NET objects with low overhead (remote objects are configured to use binary encoding for messaging).

Web services were considered briefly for this purpose, but were decided against due to the relatively higher overheads involved with XML-encoded messages, the inherent inflexibility of the HTTP protocol for the requirements at hand and the fact that each component would be required to be configured with a web services container (web server). However, web services are used for the Cross-Platform Manager's public interface since cross-platform interoperability was the primary requirement in this regard.

The objects remoted using .NET Remoting within the four distributed components of Alchemi, the Manager, Executor, Owner and Cross-Platform Manager are instances of `GManager`, `GExecutor`, `GApplication` and `CrossPlatformManager` respectively.

It should be noted that classes are named with respect to their roles vis-à-vis a grid application. This discussion therefore refers to an 'Owner' node synomymously with a 'User' node, since the node where the grid application is being submitted from can be considered to "own" the application.

"The prefix 'I' is used in type names to denote an interface, whereas 'G' is used to denote a 'grid node' class. **`GManager`**, **`GExecutor`**, **`GApplication`** derive from the **`GNode`** class which implements generic functionality for remoting the object itself and connecting to a remote Manager via the **`IManager`** interface.

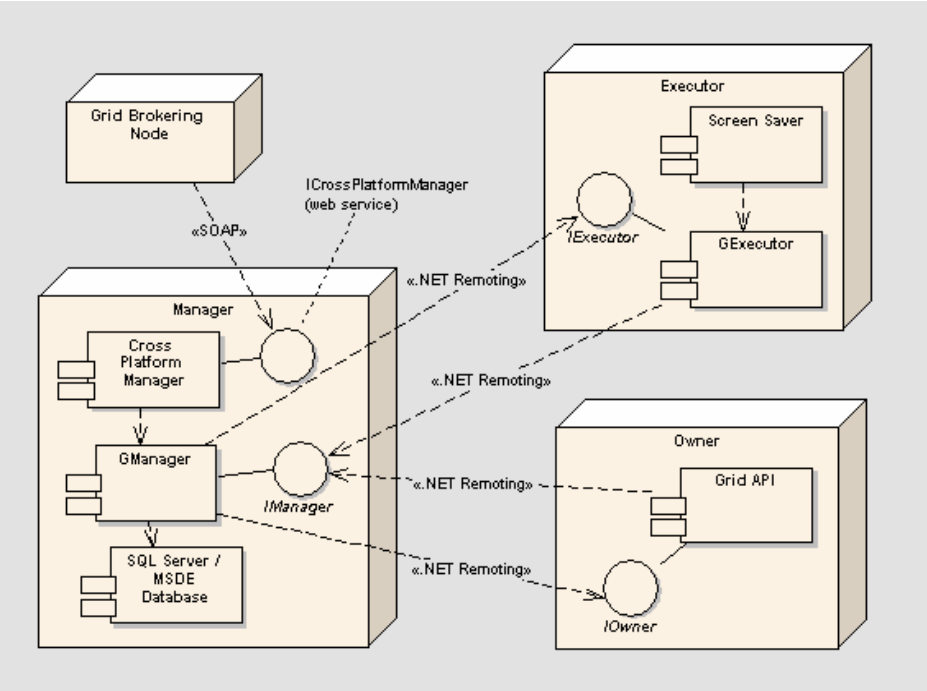

Figure 8. Alchemi architecture and interaction between its components.

The Manager executable initializes an instance of the **`GManager`** class, which is always remoted and exposes a public interface **`IManager`**. The Executor executable creates an instance of the **`GExecutor`** class. For non-dedicated execution, there is one-way communication between the Executor and the Manager. Where two-way communication is possible and dedicated execution is desired, **`GExecutor`** is remoted and exposes the **`IExecutor`** interface so that the Manager may communicate with it directly. The Executor installation provides an option to install a screen saver, which initiates non-dedicated execution when activated by the operating system.

The **`GApplication`** object in Alchemi API communicates with the Manager in a similar fashion to **`GExecutor`**. While two-way communication is currently not used in the implementation, the architecture caters for this by way of the **`IOwner`** interface.

The Cross-Platform Manager web service is a thin wrapper around **`GManager`** and uses applications and threads internally to represent tasks and jobs (the **`GJob`** class derives from **`GThread`**) via the public **`ICrossPlatformManager`** interface.

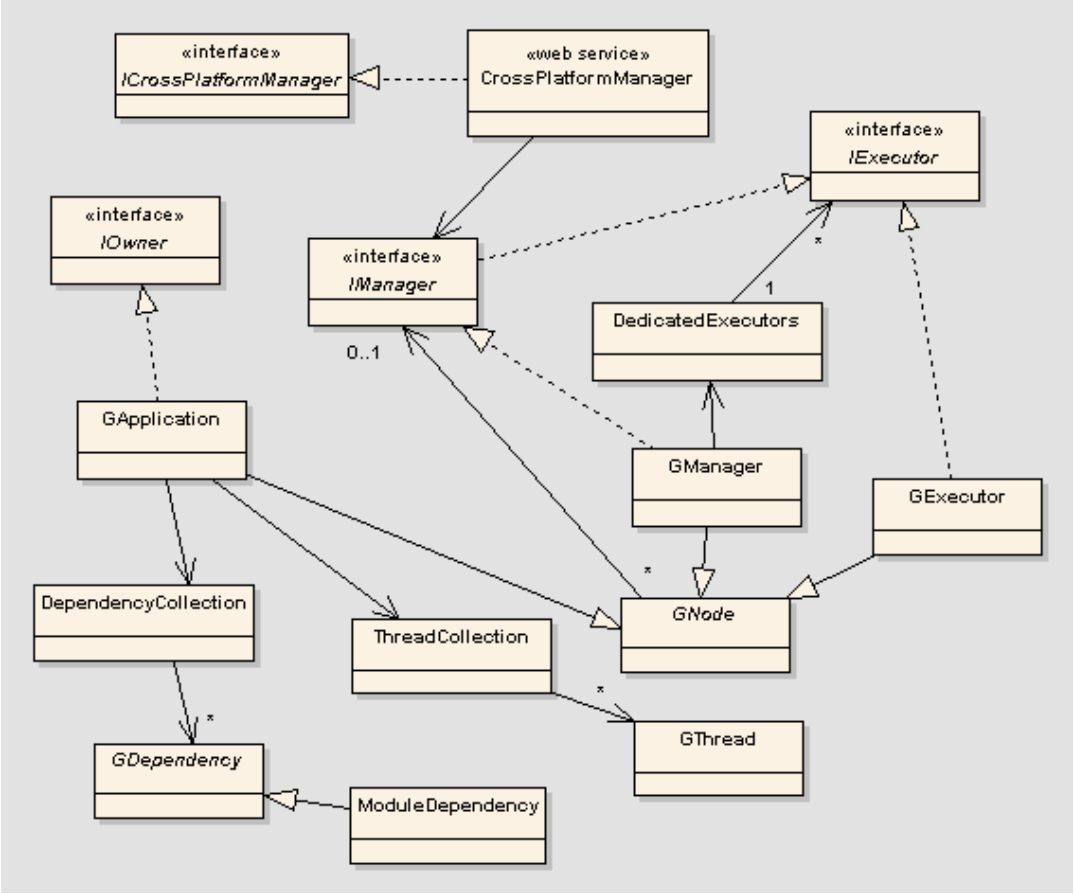

Figure 9. Main classes and their relationships.

6.2 Grid Application Lifecycle

To develop and execute a grid application the developer creates a custom grid thread class that derives from the abstract `GThread` class. An instance of the `GApplication` object is created and any dependencies required by the application are added to its `DependencyCollection`. Instances of the `GThread`-derived class are then added to the `GApplication`'s `ThreadCollection`.

The lifecycle of a grid application is shown in Figure 10 and Figure 11, showing simplified interactions between the Owner and Executor nodes respectively and the Manager.

The `GApplication` serializes and sends relevant data to the Manager, where it is persisted to disk and threads scheduled. Application and thread state is maintained in a SQL Server / MSDE database.

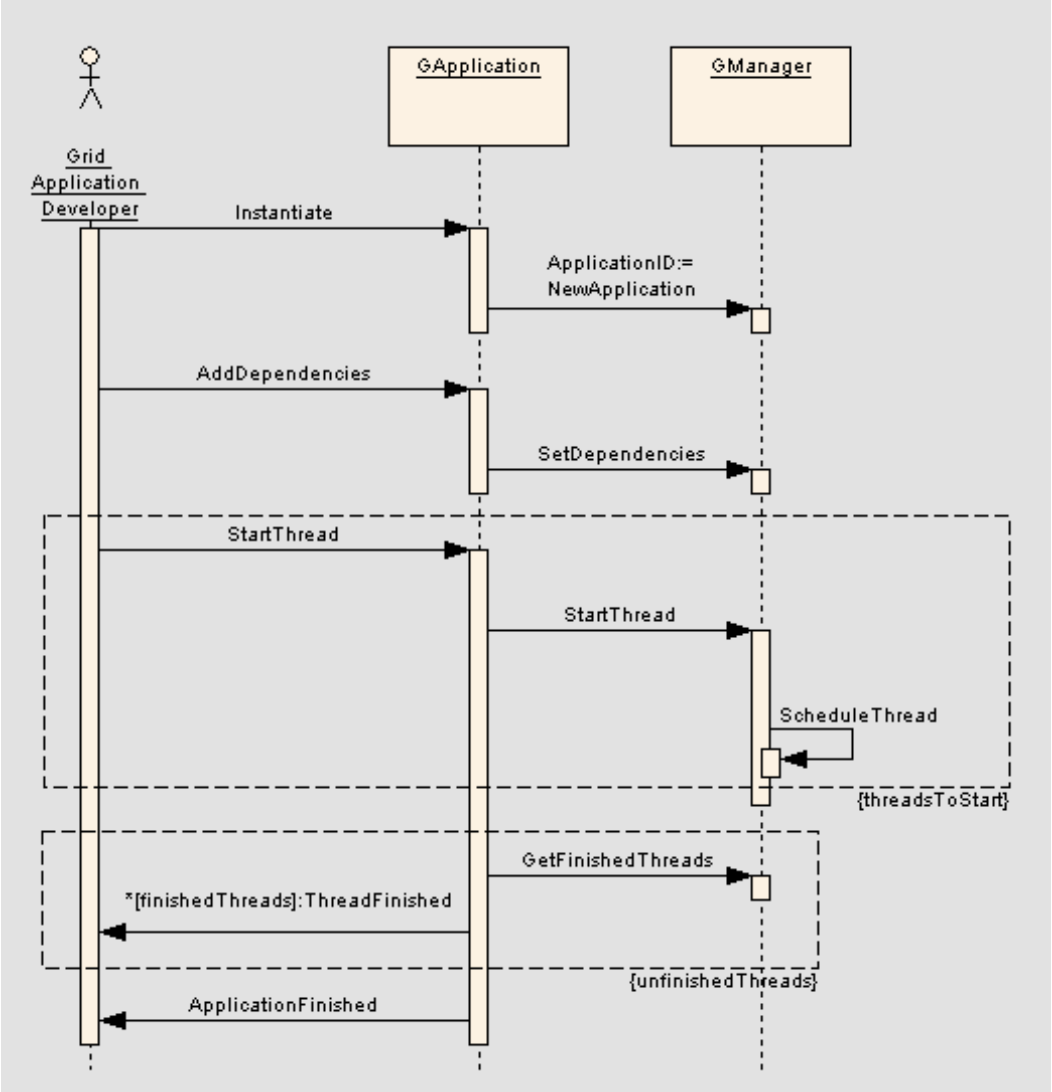

Figure 10. Simplified interaction between Owner and Manager nodes.

Non-dedicated executors poll for threads to execute until one is available. Dedicated executors are directly provided a thread to execute by the Manager.

Threads are executed in .NET application domains, with one application domain for each grid application. If an application domain does not exist that corresponds to the grid application that the thread belongs to, one is created by requesting, desterilizing and dynamically loading the application's dependencies. The thread object itself is then desterilized, started within the application domain and returned to the Manager on completion.

After sending threads to the Manager for execution, the `GApplication` polls the Manager for finished threads. A user-defined `GThreadFinish` delegate is called to signify each thread's completion and once all threads have finished a user-defined `GApplicationFinish` delegate is called.

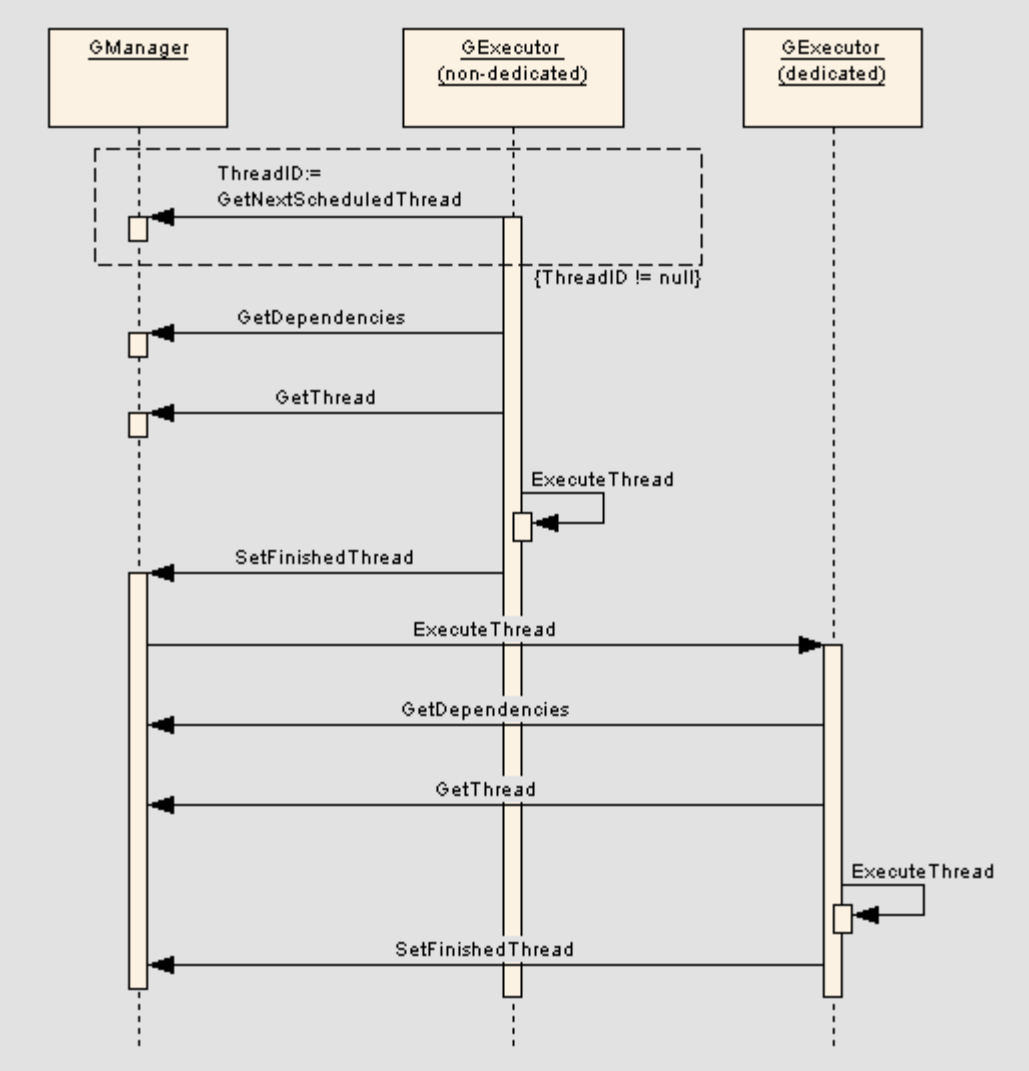

Figure 11. Simplified interaction between Executor and Manager nodes.

7 Alchemi Performance Evaluation

In this section, first we demonstrate the suitability of Alchemi to support the execution of applications created using Alchemi Grid Threads interface on a standalone desktop grid. Next, we treat an Alchemi desktop setup as one of the Grid nodes within a global Grid environment and use its job model and Web services interface to submit jobs for processing on it. This will be carried our by a Grid resource broker having an ability to interoperate with different low-level Grid middleware and schedule applications on distributed Grid nodes.

7.1 Standalone Alchemi Desktop Grid

Testbed

The testbed is an Alchemi cluster consisting of six Executors (Pentium III 1.7 GHz desktop machines with 512 MB physical memory running Windows 2000 Professional). One of these machines is additionally designated as a Manager.

Test Application & Methodology

The test application is the computation of the value of Pi to n decimal digits. The algorithm used allows the computation of the p'th digit without knowing the previous digits [29]. The application utilizes the Alchemi

grid thread model. The test was performed for a range of workloads (calculating 1000, 1200, 1400, 1600, 1800, 2000 and 2200 digits of Pi), each with one to six Executors enabled. The workload was sliced into a number of threads, each to calculate 50 digits of Pi, with the number of threads varying proportionally with the total number of digits to be calculated. Execution time was measured as the elapsed clock time for the test program to complete on the Owner node.

Results

Figure 12 shows a plot between thread size (the number of decimal places to which Pi is calculated to) and total time (in seconds taken by the all threads to complete execution) with varying numbers of Executors enabled.

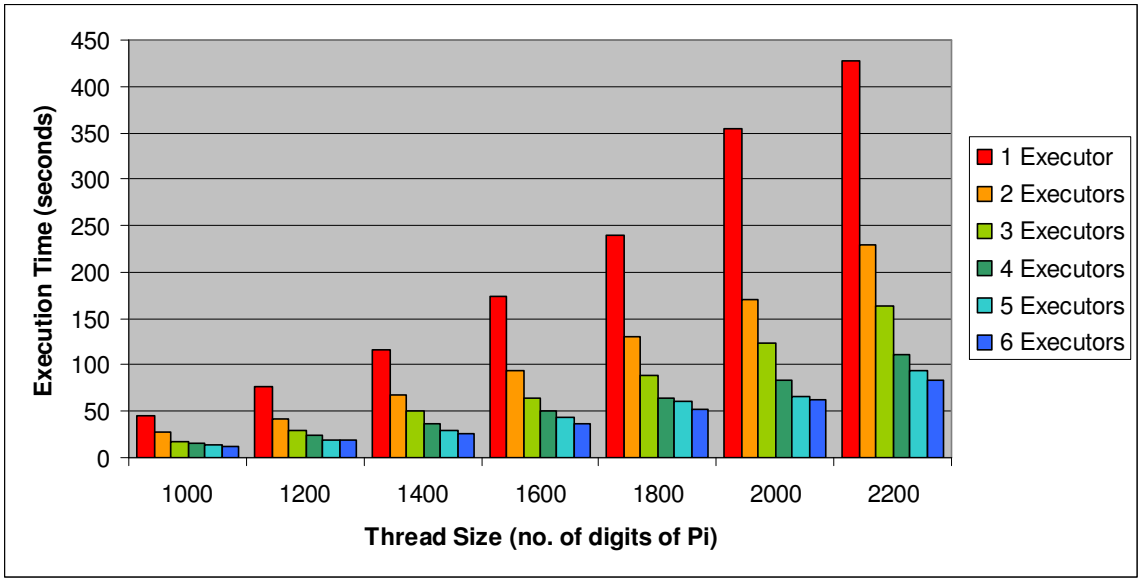

Figure 12. A plot of thread size vs. execution time on a standalone Alchemi cluster.

At a low workload (1000 digits), there is little difference between the total execution time with different quantity of Executors. This is explained by the fact that the total overhead (network latency and miscellaneous overheads involved in managing a distributed execution environment) is in a relatively high proportion to the actual total computation time. However, as the workload is increased, there is near-proportional difference when higher numbers of executors are used. For example, for 2200 digits, the execution time with six executors (84 seconds) is nearly $1/5^{th}$ of that with one executor (428 seconds). This is explained by the fact that for higher workloads, the total overhead is in a relatively lower proportion to the actual total computation time.

7.2 Alchemi as Node of a Cross-Platform Global Grid

Testbed

A global grid was used for evaluating Alchemi as a potential low-level Grid middleware with the Gridbus Grid Service Broker managing global grid resources (see Figure 13). The Windows Desktop Grid node is grid-enabled using Alchemi middleware whereas other nodes running Linux OS are Grid-enabled using Globus middleware (Globus 2.4) [7]. The Gridbus Broker developed in Java was running on Linux PC loaded with JVM (Java Virtual Machine), Globus and Alchemi Client Side Interfaces. For details on testbed setup and software confirgration see Figure 13 and Table 1. The Gridbus resource brokering mechanism obtains the users' application requirements and evaluates the suitability of various resources. It then schedules the jobs to various resources in order to satisfy those requirements.

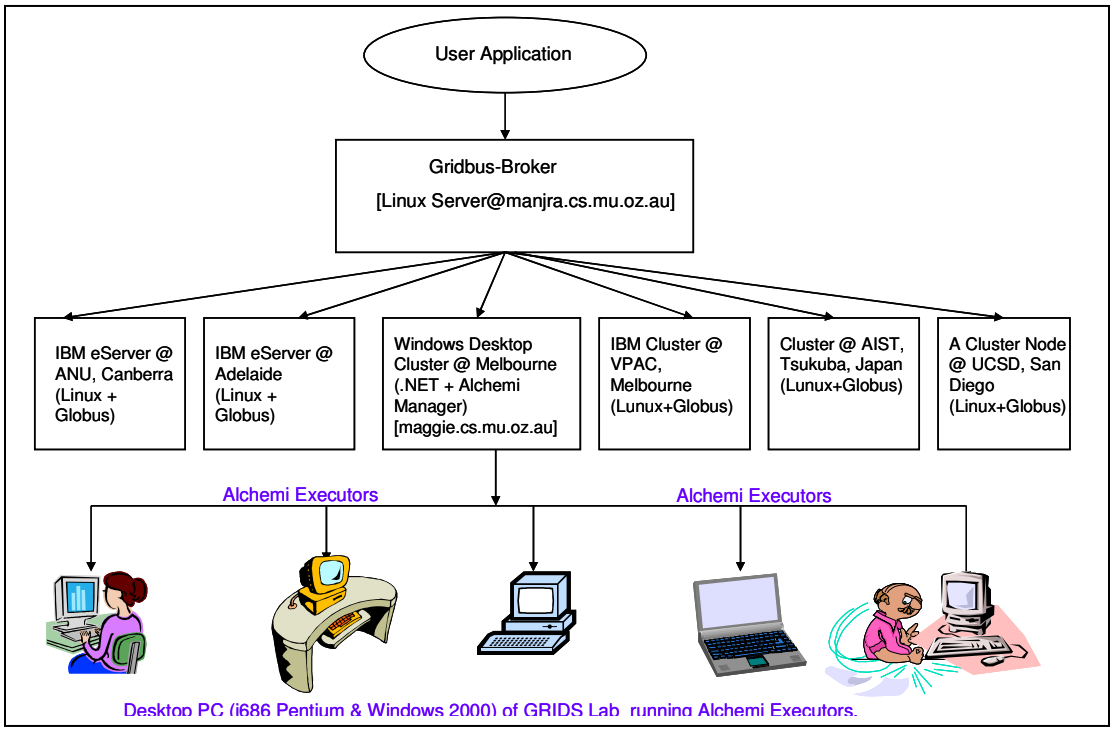

Figure 13. Testbed Setup and Software configuration.

Test Application & Methodology

For the purpose of evaluation, we used an application that calculates mathematical functions based on the values of two input parameters. The first parameter *X*, is an input to a mathematical function and the second parameter *Y,* indicates the expected calculation complexity in minutes plus a random deviation value between 0 to 120 seconds—this creates an illusion of small variation in execution time of different parametric jobs similar to a real application. A plan file modeling this application as a parameter sweep application using the Nimrod-G parameter specification language [12] is shown in Figure 14. The first part defines parameters and the second part defines the task that is to be performed for each job. As the parameter *X* varies from values 1 to 100 in step of 1, this plan file would create 100 jobs with input values from 1 to 100.

Resource	Location	Configuration	Grid Middleware	Jobs Completed
maggie.cs.mu.oz.au [Windows cluster]	University of Melbourne	6 * Intel Pentium IV 1.7 GHz	Alchemi	21
quidam.ucsd.edu [Linux cluster]	University of California, San Diego	1 * AMD Athlon XP 2100+	Globus	16
belle.anu.edu.au [Linux cluster]	Australian National University	4 * Intel Xeon 2	Globus	22
koume.hpcc.jp [Linux cluster]	AIST, Japan	4 * Intel Xeon 2	Globus	18
brecca-2.vpac.org [Linux cluster]	VPAC Melbourne	4 * Intel Xeon 2	Globus	23

Table 1. Grid resources and jobs processed.

```
#Parameter definition
parameter X integer range from 1 to 100 step 1;
parameter Y integer default 1;
#Task definition
task main
        #Copy necessary executables depending on node type
        copy calc.$OS node:calc
        #Execute program with parameter values on remote node
        node:execute ./calc $X $Y
        #Copy results file to use home node with jobname as extension
        copy node:output ./output.$jobname
endtask
```

Figure 14. Parametric job specification.

Results

The results of the experiment shown in Figure 15 show the number of jobs completed on different Grid resources at different times. The parameter calc.$OS directs the broker to select appropriate executables based a target Grid resource architecture. For example, if the target resource is Windows/Intel, it selects calc.exe and copies to the grid node before its execution. It demonstrates the feasible to utilizing Windows-based Alchemi resources along with other Unix-class resources running Globus.

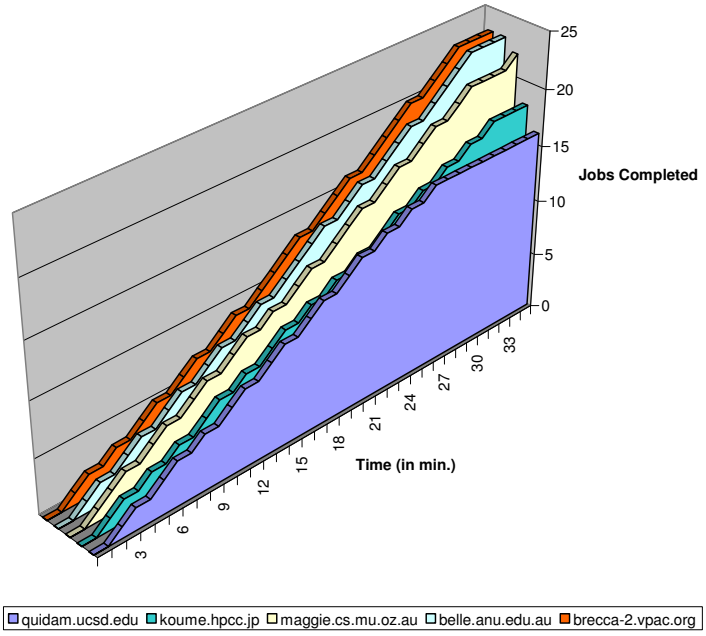

Figure 15. A plot of the number of jobs completed on different resources versus the time.

8 Summary and Future Work

We have discussed a .NET-based grid computing framework that provides the runtime machinery and object-oriented programming environment to easily construct desktop grids and develop grid applications. Its integration into the global cross-platform grid has been made possible via support for execution of grid jobs via a web services interface and the use of a broker component.

We plan to extend Alchemi in a number of areas. Firstly, support for additional functionality via the API including inter-thread communication is planned. Secondly, we are working on support for multi-clustering with peer-to-peer communication between Managers. Thirdly, we plan to support utility-based resource allocation policies driven by economic, quality of services, and service-level agreements. Fourthly, we are

investigating strategies for adherence to OGSI (Open Grid Services Infrastructure) standards by extending the current Alchemi job management interface. This is likely to be achieved by its integration with .NET-based low-level grid middleware implementations (e.g., University of Virginia's OGSI.NET [33]) that conform to grid standards such as OGSI (Open Grid Services Infrastructure) [25][32]. Finally, we plan to provide data grid capabilities to enable resource providers to share their data resources in addition to computational resources.

9 Acknowledgement and Availability

The work described in this chapter is carried as part of the Gridbus Project and is partially supported by the University of Melbourne Linkage Seed and the Australian Research Council Discovery Project grants.

Alchemi software and its documentation can be downloaded from the following web site:
http://www.alchemi.net

10 References

[1] Ian Foster and Carl Kesselman (editors), *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, USA, 1999.

[2] Larry Smarr and Charlie Catlett, Metacomputing, *Communications of the ACM Magazine*, Vol. 35, No. 6, pp. 44-52, ACM Press, USA, Jun. 1992.

[3] Microsoft Corporation, *.NET Framework Home*, http://msdn.microsoft.com/netframework/ (accessed November 2003)

[4] Piet Obermeyer and Jonathan Hawkins, *Microsoft .NET Remoting: A Technical Overview*, http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp (accessed November 2003)

[5] Microsoft Corp., *Web Services Development Center*, http://msdn.microsoft.com/webservices/ (accessed November 2003)

[6] D.H. Bailey, J. Borwein, P.B. Borwein, S. Plouffe, The quest for Pi, Math. Intelligencer 19 (1997),pp. 50-57.

[7] Ian Foster and Carl Kesselman, Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, 11(2): 115-128, 1997.

[8] Ian Foster, Carl Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International Journal of Supercomputer Applications*, 15(3), Sage Publications, 2001, USA.

[9] David Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, Dan Werthimer, SETI@home: An Experiment in Public-Resource Computing, *Communications of the ACM,* Vol. 45 No. 11, ACM Press, USA, November 2002.

[10] Yair Amir, Baruch Awerbuch, and Ryan S. Borgstrom, The Java Market: Transforming the Internet into a Metacomputer, *Technical Report CNDS-98-1*, Johns Hopkins University, 1998.

[11] Peter Cappello, Bernd Christiansen, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu, Javelin: Internet-Based Parallel Computing Using Java, *Proceedings of the 1997 ACM Workshop on Java for Science and Engineering Computation*, June 1997.

[12] Rajkumar Buyya, David Abramson, Jonathan Giddy, *Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid,* Proceedings of 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), Beijing, China, 2000.

[13] Rajkumar Buyya, Economic-based Distributed Resource Management and Scheduling for Grid Computing, *Ph.D. Thesis*, Monash University Australia, April 2002.

[14] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson, *A new major SETI project based on Project Serendip data and 100,000 personal computers*, Proceedings of the 5th International Conference on Bioastronomy, 1997.

[15] Brendon J. Wilson, *JXTA*, New Riders Publishing, Indiana, June 2002.

[16] Cecile Germain, Vincent Neri, Gille Fedak and Franck Cappello, *XtremWeb: building an experimental platform for Global Computing*, Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000), Bangalore, India, Dec. 2000.

[17] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia, Entropia: Architecture and Performance of an Enterprise Desktop Grid System, *Journal of Parallel and Distributed Computing*, Volume 63, Issue 5, Academic Press, USA, May 2003.

[18] Andy Oram (editor), *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly Press, USA, 2001.

[19] M. Litzkow, M. Livny, and M. Mutka, *Condor - A Hunter of Idle Workstations*, Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS 1988), January 1988, San Jose, CA, IEEE CS Press, USA, 1988.

[20] M. Mutka and M. Livny, The Available Capacity of a Privately Owned Workstation Environment, *Journal of Performance Evaluation, Volume 12, Issue 4,* **,** 269-284pp, Elsevier Science Publishers, The Netherlands, July 1991.

[21] N. Nisan, S. London, O. Regev, and N. Camiel, *Globally Distributed computation over the Internet: The POPCORN project*, International Conference on Distributed Computing Systems (ICDCS'98), May 26 - 29, 1998, Amsterdam, The Netherlands, IEEE CS Press, USA, 1998.

[22] Y. Aridor, M. Factor, and A. Teperman, *cJVM: a Single System Image of a JVM on a Cluster*, Proceedings of the 29th International Conference on Parallel Processing (ICPP 99), September 1999, Fukushima, Japan, IEEE Computer Society Press, USA.

[23] Intel Corporation, *United Devices' Grid MP on Intel Architecture,* http://www.ud.com/rescenter/files/wp_intel_ud.pdf (accessed November 2003)

[24] Ardaiz O., Touch J. *Web Service Deployment Using the Xbone*, Proceedings of Spanish Symposium on Distributed Systems SEID 2000.

[25] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*, January 2002.

[26] P. Cauldwell, R. Chawla, Vivek Chopra, Gary Damschen,Chris Dix, Tony Hong, Francis Norton, Uche Ogbuji, Glenn Olander, Mark A. Richman, Kristy Saunders, and Zoran Zaev, *Professional XML Web Services,* Wrox Press, 2001.

[27] E. O'Tuathail and M. Rose, Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP), *IETF RFC 3288*, June 2002.

[28] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web Services Description Language (WSDL) 1.1.W3C Note 15*, 2001. www.w3.org/TR/wsdl.

[29] World Wide Web Consortium, *XML Schema Part 0:Primer: W3C Recommendation*, May 2001.

[30] Fabrice Bellard, *Computation of the n'th digit of pi in any base in O(n^2)*, http://fabrice.bellard.free.fr/pi/pi_n2/pi_n2.html (accessed June 2003).

[31] C. Kruskal and A. Weiss, Allocating independent subtasks on parallel processors, *IEEE Transactions on Software Engineering*, 11:1001--1016, 1984.

[32] Global Grid Forum (GGF), *Open Grid Services Infrastructure (OGSI) Specification 1.0*, https://forge.gridforum.org/projects/ogsi-wg (accessed January 2004).

[33] Glenn Wasson, Norm Beekwilder and Marty Humphrey, *OGSI.NET: An OGSI-Compliant Hosting Container for the .NET Framework,* University of Virginia, USA, 2003. http://www.cs.virginia.edu/~humphrey/GCG/ogsi.net.html (accessed Jan 2004).

[34] United Devices, *The History of Distributed Computing*, http://www.ud.com/company/dc/history.htm, October 9, 2001.

[35] R. Buyya (editor), *High Performance Cluster Computing*, Vol. 1 and 2, Prentice Hall - PTR, NJ, USA, 1999.

[36] Mark Baker, Rajkumar Buyya, and Domenico Laforenza, *Grids and Grid Technologies for Wide-Area Distributed Computing*, International Journal of Software: Practice and Experience (SPE), Volume 32, Issue 15, Pages: 1437-1466, Wiley Press, USA, December 2002.

[37] R. Buyya (editor), *Grid Computing Info Centre*, http://www.gridcomputing.com/, Accessed on June 2004.

[38] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke, *Condor-G: A Computation Management Agent for Multi-Institutional Grids*, Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10), San Francisco, California, August 7-9, 2001.

Appendix A - Sample Application Employing the Grid Thread Model

```
using System;
using Alchemi.Core;

namespace Alchemi.Examples.Tutorial
{
    [Serializable]
    public class MultiplierThread : GThread
    {
        private int _A, _B, _Result;

        public int Result
```

```
        {
            get { return _Result; }
        }

        public MultiplierThread(int a, int b)
        {
            _A = a;
            _B = b;
        }

        public override void Start()
        {
            if (Id == 0) { int x = 5/Id; } // divide by zero
            _Result = _A * _B;
        }
    }

    class MultiplierApplication
    {
        static GApplication ga;

        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine("[enter] to start grid application ...");
            Console.ReadLine();

            // create grid application
            ga = new GApplication(new GConnection("localhost", 9099));

            // add GridThread module (this executable) as a dependency
            ga.Manifest.Add(new ModuleDependency(typeof(MultiplierThread).Module));

            // create and add 10 threads to the application
            for (int i=0; i<10; i++)
            {
                // create thread
                MultiplierThread thread = new MultiplierThread(i, i+1);

                // add thread to application
                ga.Threads.Add(thread);
            }

            // subscribe to events
            ga.ThreadFinish += new GThreadFinish(ThreadFinished);
            ga.ThreadFailed += new GThreadFailed(ThreadFailed);
            ga.ApplicationFinish += new GApplicationFinish(ApplicationFinished);
            // start application
            ga.Start();
            Console.ReadLine();
        }
        static void ThreadFinished(GThread th)
        {
            // cast GThread back to MultiplierThread
            MultiplierThread thread = (MultiplierThread) th;
            Console.WriteLine( "thread # {0} finished with result '{1}'",
                thread.Id, thread.Result);
        }
        static void ThreadFailed(GThread th, Exception e)
        {
            Console.WriteLine(
                "thread # {0} finished with error '{1}'", th.Id, e.Message);
        }
        static void ApplicationFinished()
        {
            Console.WriteLine("\napplication finished");
            Console.WriteLine("\n[enter] to continue ...");
        }
    }
}
```