



Alchemi: A .NET-based Enterprise Grid System and Framework

User Guide for Alchemi 1.0

December 2005

Authors: Krishna Nadiminti, Akshay Luther, Rajkumar Buyya



**Grid Computing and Distributed Systems (GRIDS) Laboratory
Dept. of Computer Science and Software Engineering
The University of Melbourne, Australia**

Table of Contents

1. Introduction and Concepts.....	3
1.1. The Network is the Computer	3
1.2. How Alchemi Works.....	3
2. Installation, Configuration and Operation	4
2.1. Common Requirements	4
2.2. Manager	4
2.3. Role-Based Security	7
2.4. Cross Platform Manager	7
2.5. Executor	8
2.6. Software Development Kit.....	11
3. Grid Programming.....	13
3.1. Introduction to Grid Software	13
3.2. Grid Programming Tutorial.....	15
3.3. Other Features of the Alchemi API	18
3.4. Grid Job Model : Grid-Enabling Legacy Applications	20
3.5. Drag & Drop Components.....	22
3.6. Cross-Platform Development: The Cross-Platform Manager Web Service.....	23
4. Conclusion	23
5. Alchemi References	24

Abstract: This document is a guide to using the Alchemi .NET Grid Computing System and Framework. Please mail your suggestions, clarifications, comments, and your usage details to Krishna Nadiminti (kna@cs.mu.oz.au).

1. Introduction and Concepts

This section gives you an introduction to how Alchemi implements the concept of grid computing and discusses concepts required for using Alchemi. Some key features of the framework are highlighted along the way.

1.1. The Network is the Computer

The idea of meta-computing - the use of a network of many independent computers as if they were one large parallel machine, or virtual supercomputer - is very compelling since it enables supercomputer-scale processing power to be had at a fraction of the cost of traditional supercomputers.

While traditional virtual machines (e.g. clusters) have been designed for a small number of tightly coupled homogeneous resources, 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 in corporate and home environments are heavily underutilized – typically only one-tenth of processing power is used – has given rise to interest in harnessing the vast amounts of processing power that is available in the form of spare CPU cycles on Internet- or intranet-connected desktops. This new paradigm has been dubbed Grid Computing.

Popular distributed computing projects that proved the feasibility of the concept are distributed.net and Seti @ Home.

1.2. How Alchemi Works

There are four types of distributed components (nodes) involved in the construction of Alchemi grids and execution of grid applications: Manager, Executor, User & Cross-Platform Manager.

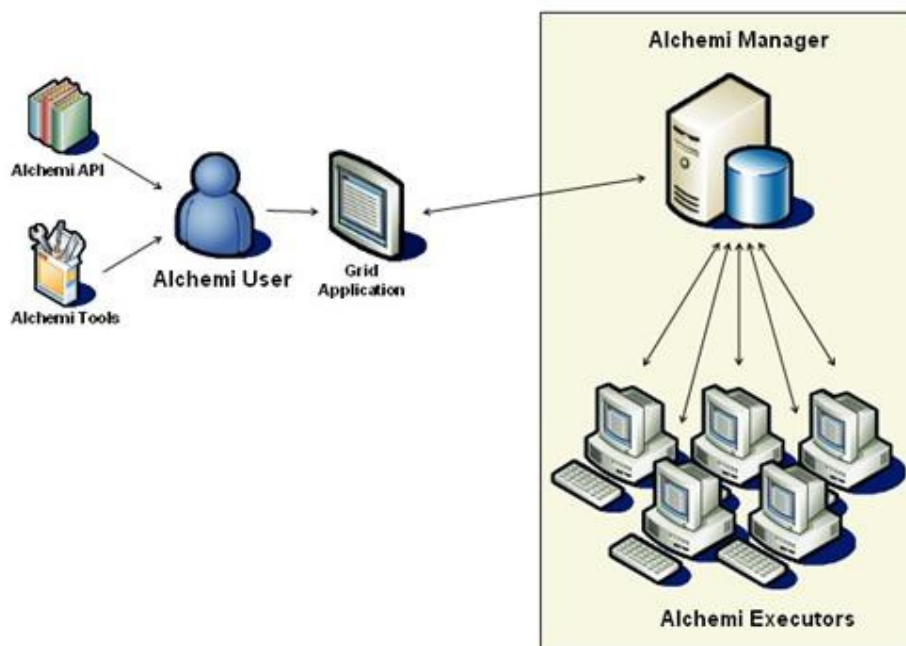


Figure 1. An Alchemi Grid

A grid is created by installing Executors on each machine that is to be part of the grid and linking them to a central Manager component. The Windows installer setup that comes with the Alchemi distribution and minimal configuration makes it very easy to set up a grid.

An Executor can be configured to be dedicated (meaning the Manager initiates thread execution directly) or non-dedicated (meaning that thread execution is initiated by the Executor.) Non-dedicated

Executors can work through firewalls and NAT servers since there is only one-way communication between the Executor and Manager. Dedicated Executors are more suited to an intranet environment and non-dedicated Executors are more suited to the Internet environment.

Users can develop, execute and monitor grid applications using the .NET API and tools which are part of the Alchemi SDK. Alchemi offers a powerful grid thread programming model which makes it very easy to develop grid applications and a grid job model for grid-enabling legacy or non-.NET applications.

An optional component (not shown) is the Cross Platform Manager web service which offers interoperability with custom non-.NET grid middleware.

2. Installation, Configuration and Operation

This section documents the installation, configuration and operation of the various parts of the framework for setting up Alchemi grids. The various components can be downloaded from:

<http://www.alchemi.net/download.html>

2.1. Common Requirements

- Microsoft .NET Framework 1.1

2.2. Manager

The Manager should be installed on a stable and reasonably capable machine. The Manager requires:

- SQL Server 2000 or MSDE 2000

If using SQL Server, ensure that SQL Server authentication is enabled. Otherwise, follow these instructions to install and prepare MSDE 2000 for Alchemi. Make a note of the system administrator (sa) password in either case. [Note: SQL Server / MSDE do not necessarily need to be installed on the same machine as the Manager.]

Installation

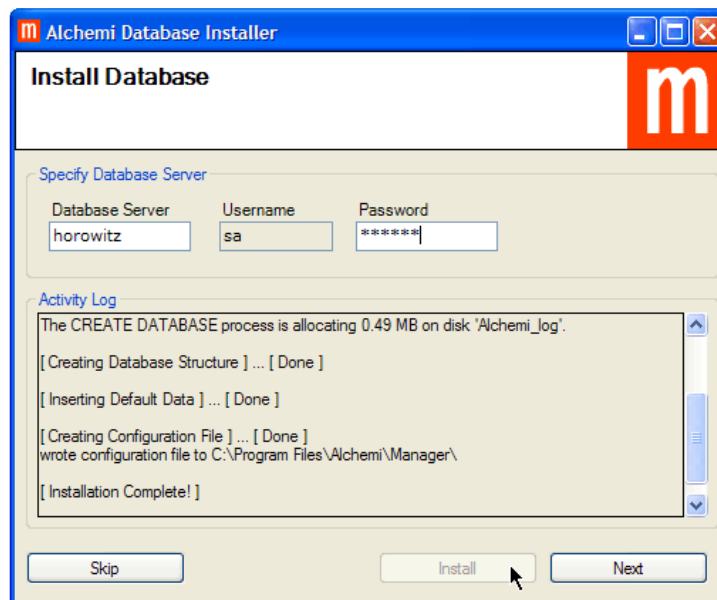


Figure 2. Alchemi Database installation

- The Alchemi Manager can be installed in two modes
 - As a normal Windows desktop application
 - As a windows service. (supported only on Windows NT/2000/XP/2003)
- To install the manager as a windows application, use the Manager Setup installer. For service-mode installation use the Manager Service Setup. The configuration steps are the same for both modes. In case of the service-mode, the “Alchemi Manager Service” installed and configured to run automatically on Windows start-up. After installation, the standard Windows service control manager can be used to control the service. Alternatively the Alchemi ManagerServiceController program can be used. The Manager service controller is a graphical interface, which is exactly similar to the normal Manager application.
- Install the Manager via the Manager installer. Use the sa password noted previously to install the database during the installation.

Configuration & Operation

- The Manager can be run from the desktop or Start -> Programs -> Alchemi -> Manager -> Alchemi Manager. The database configuration settings used during installation automatically appear when the Manager is first started.
- Click the "Start" button to start the Manager.
- When closed, the Manager is minimised to the system tray.

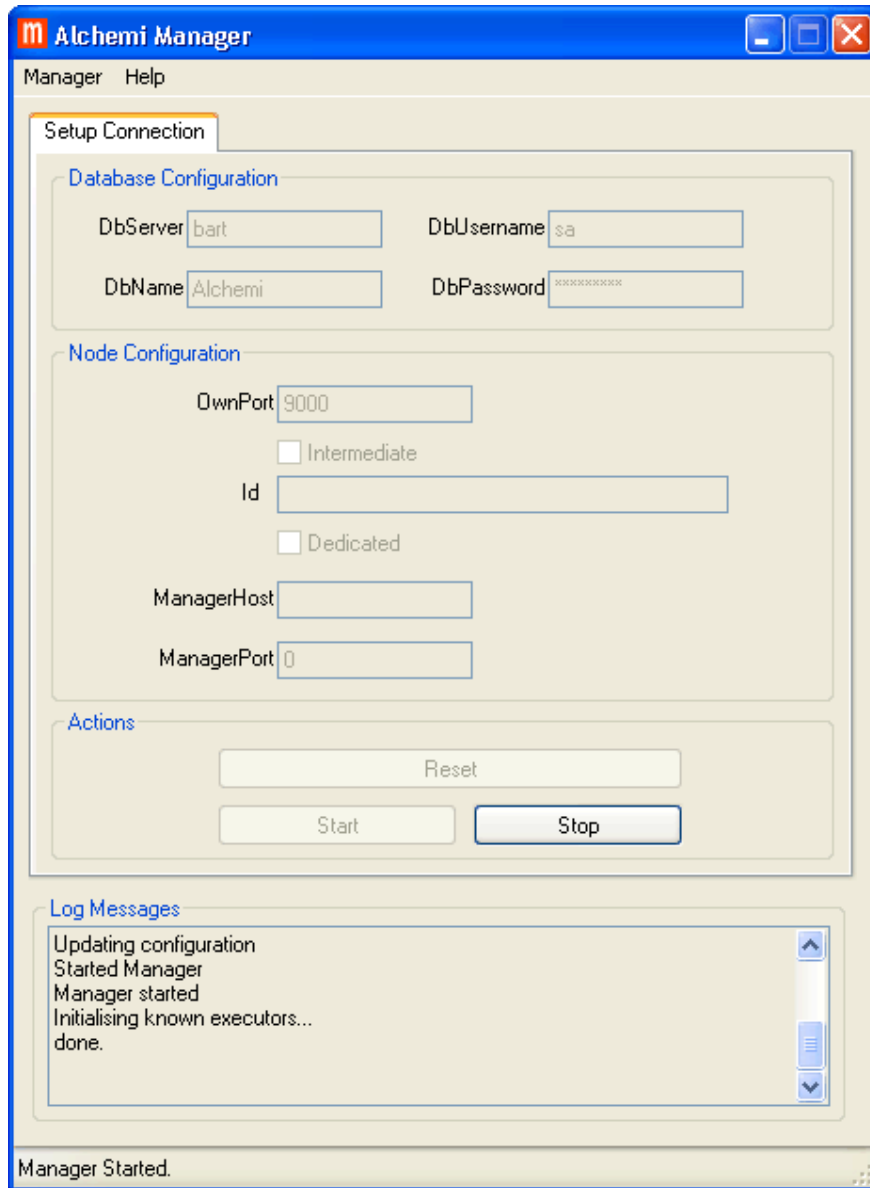


Figure 3. Manager operation.



Figure 4. Manager minimised to system tray.

Under service-mode operation, the GUI shown in fig. 3 is used to start / stop the Manager service. The service will continue to operate even after the service controller application exits.

Manager Logging

The manager logs its output and errors to a log file called "alchemi-manager.log". This can be used to debug the manager / report errors / verify the manager operation. The log file is placed in the "dat" directory under the installation directory.

2.3. Role-Based Security

Every program connecting to the Manager must supply a valid username and password. Three default accounts are created during installation: executor (password: executor), user (password: user) and admin (password: admin) belonging to the 'Executors', 'Users' and 'Administrators' groups respectively.

Users are administered via the 'Users' tab of the Alchemi Console (located in the Alchemi SDK). Only Administrators have permissions to manage users; you must therefore initially log in with the default admin account.

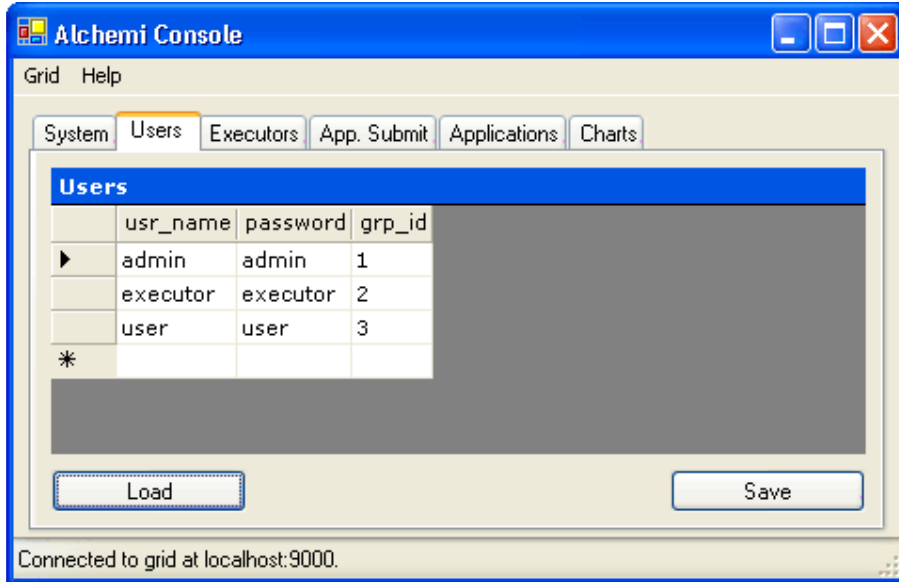


Figure 5. User administration via the Alchemi Console (showing default accounts).

The Console lets you add users, modify their group membership and change passwords.

The Users group (grp_id = 3) is meant for users executing grid applications.

The Executors group (grp_id = 2) is meant for Alchemi Executors. By default, Executors attempting to connect to the Manager will use the executor account. If you do not wish Executors to connect anonymously, you can change the password for this account.

You should change the default admin password for production use.

2.4. Cross Platform Manager

The Cross Platform Manager (XPManager) requires:

- Internet Information Services (IIS)
- ASP.NET

Installation

- Install the XPManager web service via the Cross Platform Manager installer.

Configuration

- If the XPManager is installed on a different machine than the Manager, or if the default port of the Manager is changed, the web service's configuration must be modified. The XPManager is configured via the ASP.NET Web.config file located in the installation directory (wwwroot\Alchemi\CrossPlatformManager by default):

```
<appSettings>
  <add key="ManagerUri" value="tcp://localhost:9000/Alchemi_Node" />
</appSettings>
```

Operation

- The XPManager web service URL is of the format
`http://[host_name]/[installation_path]`
- The default is therefore
`http://[host_name]/Alchemi/CrossPlatformManager`
- The web service interfaces with the Manager. The Manager must therefore be running and started for the web service to work.

2.5. Executor

Installation

- The Alchemi Executor can be installed in two modes
 - As a normal Windows desktop application
 - As a windows service. (supported only on Windows NT/2000/XP/2003)
- To install the executor as a windows application, use the Executor Setup installer. For service-mode installation use the Executor Service Setup. The configuration steps are the same for both modes. In case of the service-mode, the "Alchemi Executor Service" installed and configured to run automatically on Windows start-up. After installation, the standard Windows service control manager can be used to control the service. Alternatively the Alchemi ExecutorServiceController program can be used. The Executor service controller is a graphical interface, which looks very similar to the normal Executor application.
- Install the Executor via the Executor installer and follow the on-screen instructions.

Configuration & Operation

The Executor can be run from the desktop or Start -> Programs -> Alchemi -> Executor -> Alchemi Executor.

The Executor is configured from the application itself.

You need to configure 2 aspects of the Executor:

- The host and port of the Manager to connect to.
- Dedicated / non-dedicated execution. A non-dedicated Executor executes grid threads on a voluntary basis (it requests threads to execute from the Manager), while a dedicated Executor is always executing grid threads (it is directly provided grid threads to execute by the Manager). A non-dedicated Executor works behind firewalls.
- Click the "Connect" button to connect the Executor to the Manager.

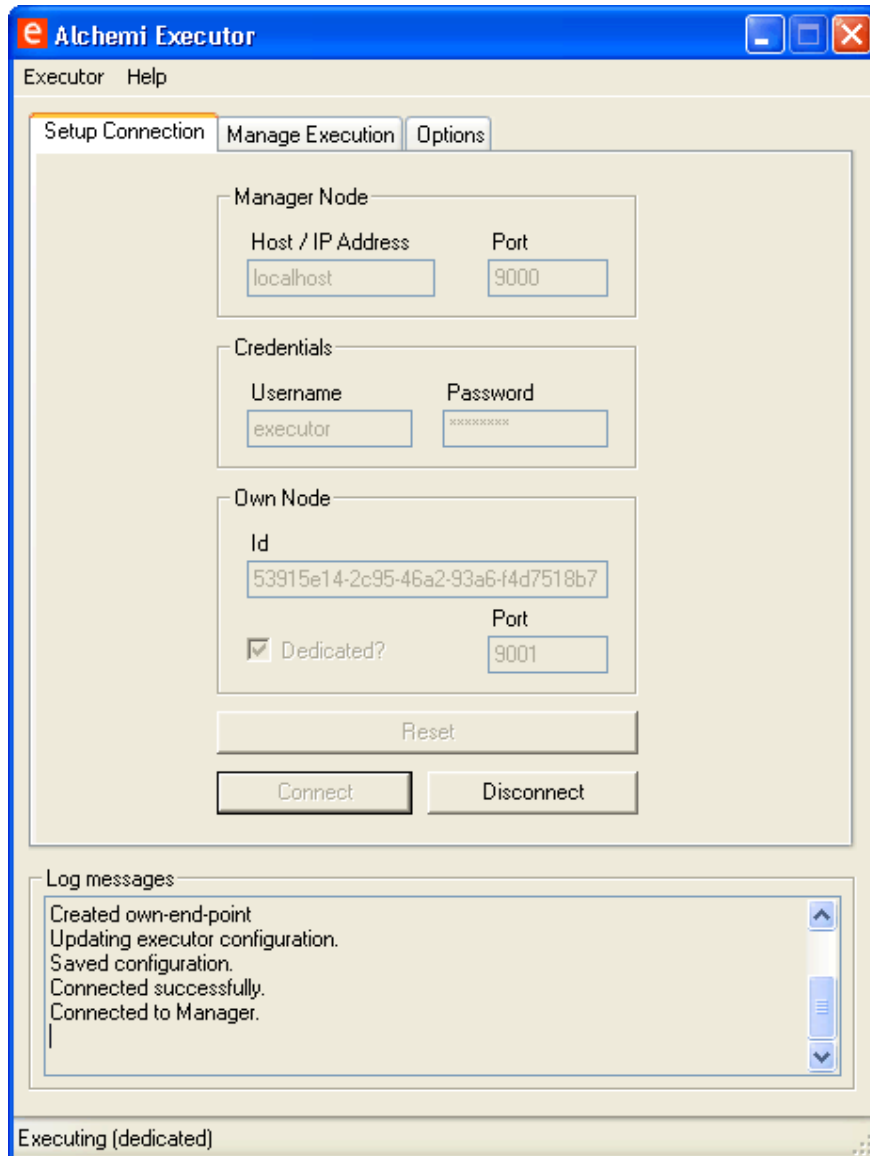


Figure 6. Executor connected to a Manager.

- If the Executor is configured for non-dedicated execution, you can start executing by clicking the "Start Executing" button in the "Manage Execution" tab.

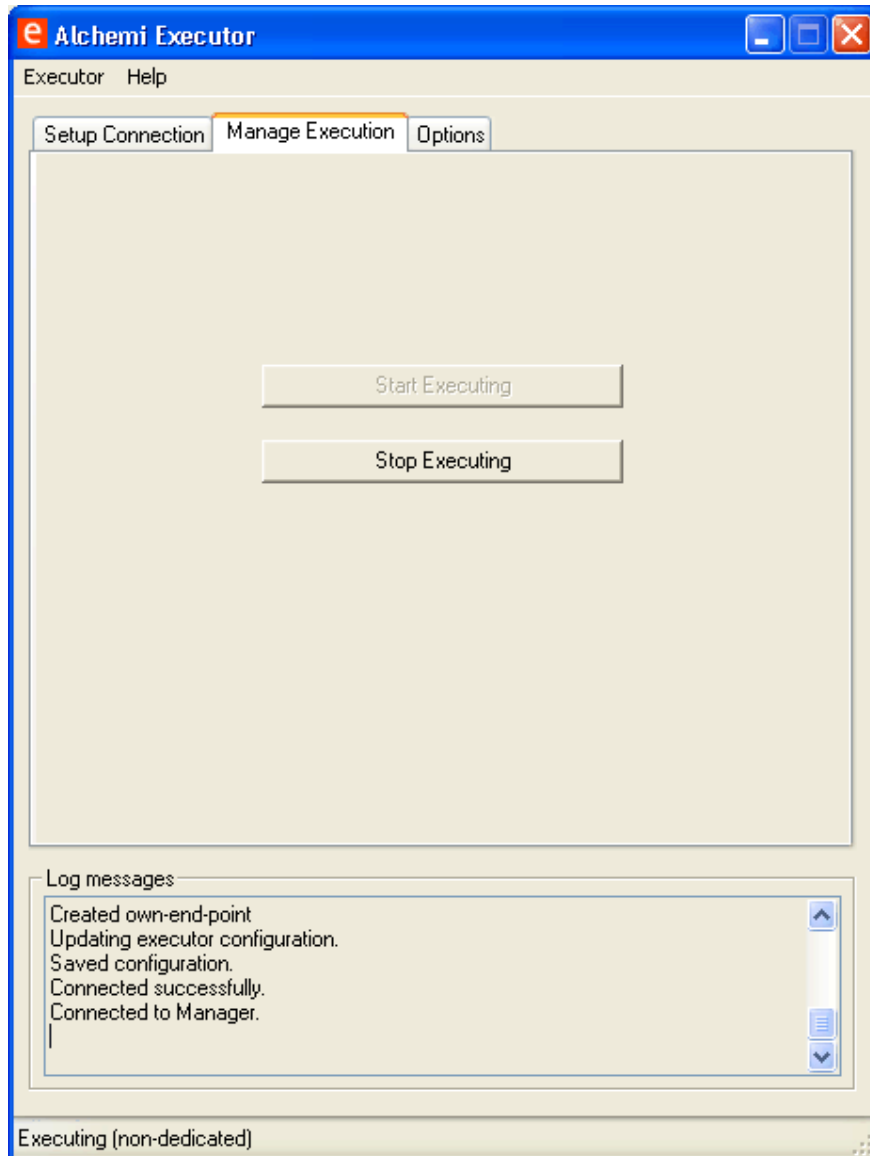


Figure 7. Non-dedicated execution.

The Executor only utilises idle CPU cycles on the machine and does not impact on the CPU usage of running programs. When closed, the Executor sits in the system tray. Other options such as a interval of executor heartbeat (i.e time between pinging the Manager) can be configured via the options tab.

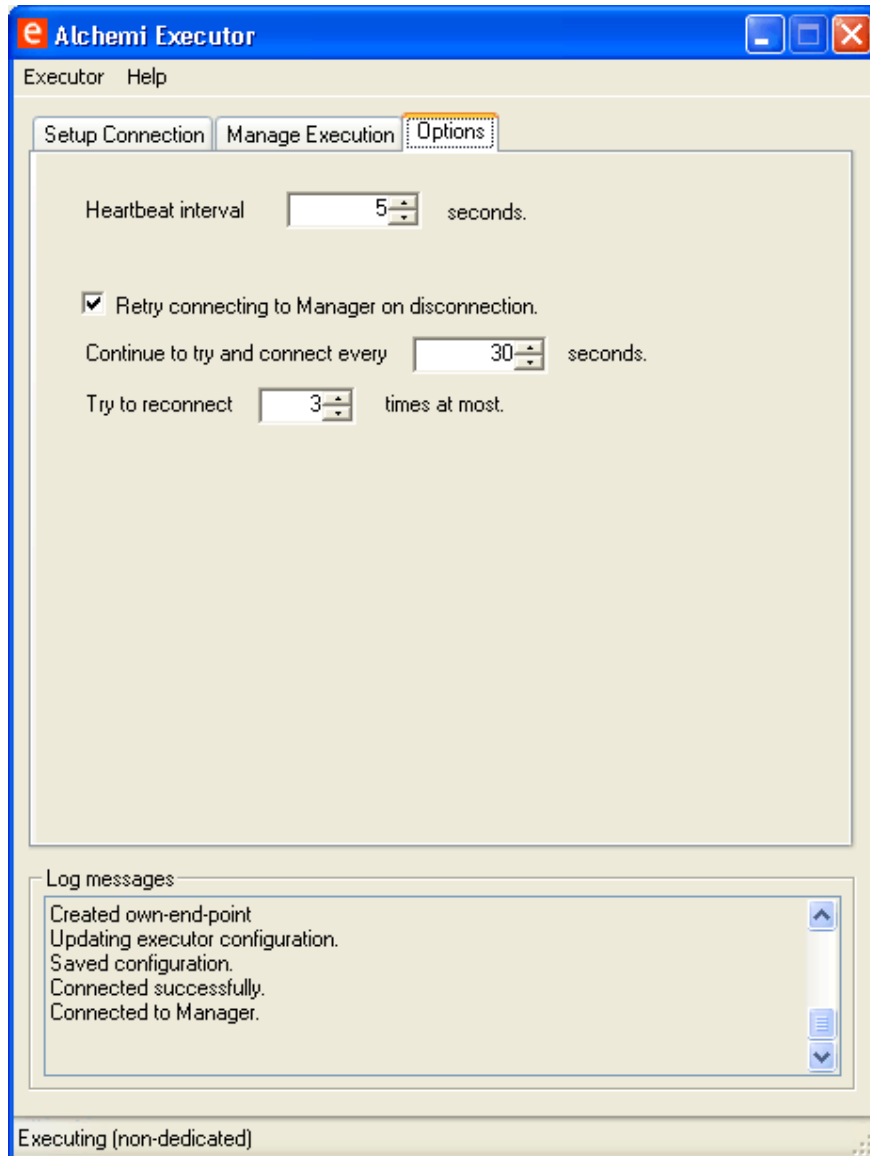


Figure 8. Additional Executor Options



Figure 9. Executor minimised to system tray.

Under service-mode operation, the GUI shown in fig. 8 is used to start / stop the Executor service. The service will continue to operate even after the service controller application exits.

Executor Logging

The executor logs its output and errors to a log file called "alchemi-executor.log". This can be used to debug the executor / report errors / verify the executor operation. The log file is placed in the "dat" directory under the installation directory.

2.6. Software Development Kit

The SDK can be unzipped to a convenient location. It contains the following:

Alchemi Console

The Console (Alchemi.Console.exe) is a grid administration and monitoring tool. It is located in the bin directory.

The 'Summary' table shows system statistics and a real-time graph of power availability and usage. The 'Applications' tab lets you monitor running applications. The 'Executors' tab provides information on Executors. The 'Users' tab lets you manage users (see 2.3. Role-Based Security).

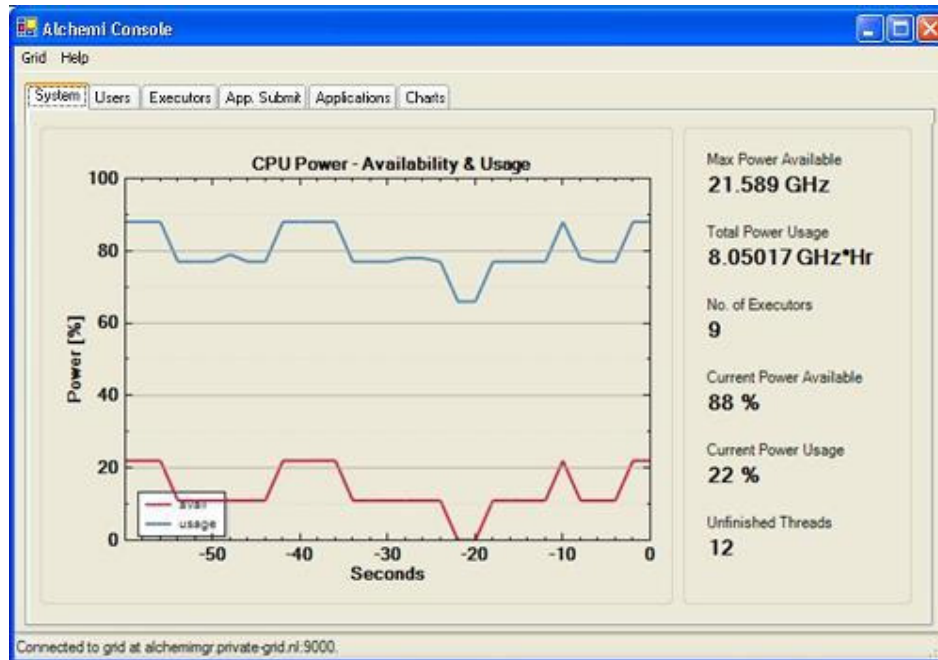


Figure 10. Console 'System' tab.

The screenshot shows the 'Applications' tab of the Alchemi Console. It displays a table of applications and a table of jobs. Below the application table are 'Load' and 'Stop' buttons. Below the jobs table is a 'Stop' button.

application_id	time_created	is_primary	usr_name	application_n
3526619f-be29-4686-9f63-05e6d59e6d4e	27/05/2005 5:34 PM	True	user	(null)
cfca5041-df2a-4afa-b172-0acbee7ea3ed	1/06/2005 3:40 PM	True	admin	(null)
f04a25f0-aae3-4f43-9a57-188cf3e303e5	27/05/2005 8:03 PM	True	user	(null)
0607d9ae-9244-4f43-b662-2dac39b02c15	17/06/2005 12:41 PM	True	user	(null)
337bf9dd-61e7-40a3-b3cd-414faecbdb09	17/06/2005 10:57 AM	True	user	(null)
839ce8dh-c03f-4a8c-h5h2-44afa333ha79	26/05/2005 4:42 PM	True	user	(null)

thread_id	time_started	time_finished	executor_id	priority	failed	Sts
0	27/05/2005 8:03 PM	27/05/2005 8:03 PM	53915e14-2c95-46a2-93a6-f4d7518b7eae	5	False	De
1	27/05/2005 8:03 PM	27/05/2005 8:03 PM	53915e14-2c95-46a2-93a6-f4d7518b7eae	5	False	De
2	27/05/2005 8:03 PM	27/05/2005 8:03 PM	53915e14-2c95-46a2-93a6-f4d7518b7eae	5	False	De
3	27/05/2005 8:03 PM	27/05/2005 8:03 PM	53915e14-2c95-46a2-93a6-f4d7518b7eae	5	False	De
4	27/05/2005 8:03 PM	27/05/2005 8:03 PM	53915e14-2c95-46a2-93a6-f4d7518b7eae	5	False	De

At the bottom of the window, it says 'Connected to grid at localhost:9000.'

Figure 11. Console 'Applications' tab.

Alchemi.Core.dll

Alchemi.Core.dll is a class library for creating grid applications to run on Alchemi grids. It is located in the bin directory. It must be referenced from by all your grid applications. (For more on developing grid applications, please see section 3. Grid Programming).

Examples

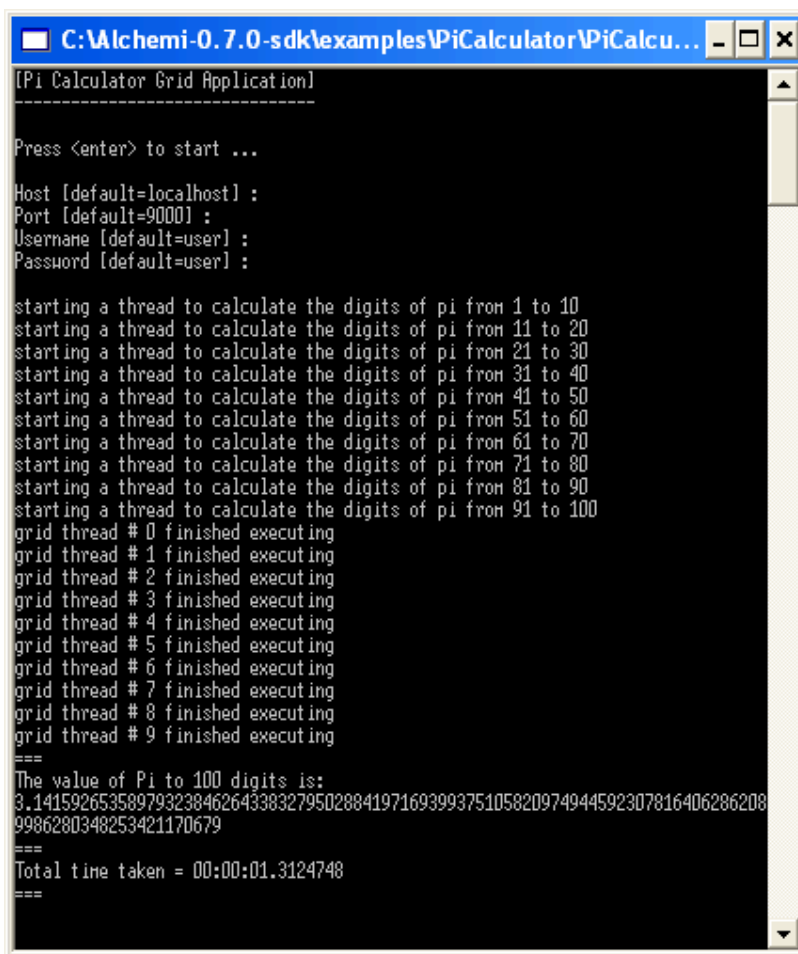
The examples directory contains several example grid applications that let you test grid creation and demonstrate grid programming.

2.6. Verifying Grid Installation

You can verify successful setup of a grid by running a sample application on it. The examples directory in the SDK contains a number of sample applications.

One such application is "PiCalculator" that calculates the value of Pi to 100 digits:

- SDK\examples\PiCalculator\PiCalculatorExec\bin\debug\PiCalculator.exe



```
[Pi Calculator Grid Application]
-----
Press <enter> to start ...

Host [default=localhost] :
Port [default=9000] :
Username [default=user] :
Password [default=user] :

starting a thread to calculate the digits of pi from 1 to 10
starting a thread to calculate the digits of pi from 11 to 20
starting a thread to calculate the digits of pi from 21 to 30
starting a thread to calculate the digits of pi from 31 to 40
starting a thread to calculate the digits of pi from 41 to 50
starting a thread to calculate the digits of pi from 51 to 60
starting a thread to calculate the digits of pi from 61 to 70
starting a thread to calculate the digits of pi from 71 to 80
starting a thread to calculate the digits of pi from 81 to 90
starting a thread to calculate the digits of pi from 91 to 100
grid thread # 0 finished executing
grid thread # 1 finished executing
grid thread # 2 finished executing
grid thread # 3 finished executing
grid thread # 4 finished executing
grid thread # 5 finished executing
grid thread # 6 finished executing
grid thread # 7 finished executing
grid thread # 8 finished executing
grid thread # 9 finished executing
===
The value of Pi to 100 digits is:
3.141592653589793238462643383279502884197169399375105820974944592307816406286208
9986280348253421170679
===
Total time taken = 00:00:01.3124748
===
```

Figure 12. PiCalculator running on a grid.

3. Grid Programming

This section is a guide to developing Alchemi grid applications.

3.1. Introduction to Grid Software

For the purpose of grid application development, a grid can be viewed as an aggregation of multiple machines (each with one or more CPUs) abstracted to behave as one "virtual" machine with multiple

CPUs. However, grid implementations differ in the way they implement this abstraction and one of the key differentiating features of Alchemi is the way it abstracts the grid, with the aim to make the process of developing grid software as easy as possible.

Due to the nature of the grid environment (loosely coupled, heterogenous resources connected over an unreliable, high-latency network), grid applications have the following features:

- They can be parallelised into a number of independent computation units
- Work units have a high computation time vs. communication time ratio

Alchemi supports two models for parallel application composition.

Course-Grained Abstraction: File-Based Jobs

Traditional grid implementations have only offered a high-level abstraction of the virtual machine, where the smallest unit of parallel execution is a process. The specification of a job to be executed on the grid at the most basic level consists of input files, output files and an executable (process). In this scenario, writing software to run on a grid involves dealing with files, an approach that can be complicated and inflexible.

Fine-Grained Abstraction: Grid Threads

On the other hand, the primary programming model supported by Alchemi offers a more low-level (and hence more powerful) abstraction of the underlying grid by providing a programming model that is object-oriented and that imitates traditional multi-threaded programming.

The smallest unit of parallel execution in this case is a grid thread (.NET object), where a grid thread is programmatically analogous to a "normal" thread (without inter-thread communication).

The grid application developer deals only with grid thread and grid application .NET objects, allowing him/her to concentrate on the application itself without worrying about the "plumbing" details. Furthermore, abstraction at this level allows the use of a elegant programming model with clean interfacing between remote and local code.

Note: Hereafter, applications and threads can be taken to mean grid applications and grid threads respectively, unless stated otherwise.

Grids Jobs vs. Grid Threads

Support for execution of grid jobs (programmatically as well as declaratively) is present for the following reasons:

- Grid-enabling legacy or non-.NET applications
- Interoperability with grid middleware on other platforms (via a web services interface)

The grid thread model is preferred due to its ease of use, power and flexibility and should be used for new applications, while the grid job model should be used for grid-enabling legacy/non-.NET applications or by non-.NET middleware interoperating with Alchemi.

Traditional Multiple-Processor Paradigm vs. Grid Paradigm

To clarify the concepts discussed so far, consider the following analogies between traditional multithreaded programming and grid programming:

Multiprocessor Machine	Grid
Operating System	Alchemi
Processor	Executor
OS Services (High-Level) - System API	Alchemi API

OS Services (Low-Level)	Manager
Process	Grid Application
Thread	Grid Thread

3.2. Grid Programming Tutorial

This tutorial provides a basic introduction to grid programming with Alchemi. Familiarity with VS.NET and C# is assumed.

We will write a simple grid application that generates prime numbers. This involves generating some large numbers at random and using a grid to check whether they are prime or not.

Follow these steps to set up a development environment:

- Construct a minimal grid (1 Manager and 1 Executor) on the development machine (see 2. Installation, Configuration and Operation) and test it by running PiCalculator (see 2.6. Verifying Grid Installation)
- Download the Alchemi SDK and extract to a convenient location
- Locate Alchemi.Core.dll for referencing in applications

As discussed, an Alchemi grid can be viewed as a virtual machine with multiple processors. A grid application can take advantage of this by creating independent units of work to be executed in parallel on the grid (each unit of work is executed by a particular Executor).

These units of work are called grid threads and must be instances of a class that is derived from Alchemi.Core.Owner,GThread. Code that is to be executed on the grid is defined in this class's void Start() method.

To write a grid thread class, we derive a new class from the Alchemi.Core. Owner,GThread class and override the void Start() method. We must also add the Serializable attribute to it:

```
[Serializable]
class CustomGridThread : GThread
{
    public override void Start()
    {
    }
}
```

The process of checking whether a number is prime or not, is computationally intensive. Also, the process can be carried out for several numbers in parallel. It can thus be implemented as a grid thread.

To start off,

- Create a new console application project, "PrimeNumberGenerator"
- Add a reference to Alchemi.Core.dll (Alchemi.Core.dll must be referenced by any projects using the Alchemi API)

We now write a grid thread that checks whether a number is prime or not. We will use a naive algorithm that counts the number of factors of the number between 1 and the number itself. If the number of factors is 2 (1 and the number itself), then by definition, the number is prime.

```
using System;
using Alchemi.Core;

namespace Tutorial
{
```

```

[Serializable]
class PrimeNumberChecker : GThread
{
    public readonly int Candidate;
    public int Factors = 0;

    public PrimeNumberChecker(int candidate)
    {
        Candidate = candidate;
    }

    public override void Start()
    {
        // count the number of factors of the number from 1 to the
number itself
        for (int d=1; d<=Candidate; d++)
        {
            if (Candidate%d == 0) Factors++;
        }
    }
}
...

```

Note: There are more efficient algorithms to check whether a number is prime is not. However, the algorithm itself is not our concern; the purpose of this tutorial is to teach parallel programming in a grid environment using Alchemi.

Now that we have written our parallel code, we can go about writing code that runs it on the grid. For this, we use the GApplication class.

```

...

class PrimeNumberGenerator
{
    public static GApplication App = new GApplication();

    [STAThread]
    static void Main(string[] args)
    {
        // create grid threads to check if some randomly generated
large
        // numbers are prime
        Random rnd = new Random();
        for (int i=0; i<100; i++)
        {
            App.Threads.Add(new
PrimeNumberChecker(rnd.Next(1000000)));
        }

        // initialise application
        Init();

        // start the application
        App.Start();

        Console.ReadLine();

        // stop the application
        App.Stop();
    }
}

```

```
...
```

GApplication is a container for executing grid threads and is used to interact with the grid. In the Main method of the application, 100 PrimeNumberChecker grid threads are created and added to the App grid application's Threads property (a GThread collection).

The application is initialised by calling the Init() method (discussed next) and execution of all threads added to it is initiated by calling its Start() method.

```
...  
  
private static void Init()  
{  
    // specify connection properties  
    App.Connection = new GConnection("localhost", 9000,  
"user", "user");  
  
    // grid thread needs to  
    App.Manifest.Add(new  
ModuleDependency(typeof(PrimeNumberChecker).Module));  
  
    // subscribe to ThreadFinish event  
    App.ThreadFinish += new GThreadFinish(App_ThreadFinish);  
}  
  
...
```

Before an application can be started (and threads executed on a grid), some grid connection properties must be specified (via the GConnection class) and set (via the application's Connection property). These are:

- Manager host
- Manager port
- Username
- Password

For more information on Alchemi security, please see 2.3. Role-Based Security.

The Manifest property of GApplication lets you specify dependencies for the grid threads in the application. Dependencies for a particular application are dynamically deployed to an Executor when it executes a thread belonging to an application for the first time. A necessary dependency to be added to the application's manifest is the file containing the implementation of your grid thread. This must be specified so that an Executor can recreate the thread and execute it. The ModuleDependency class lets you specify a particular module (.dll or .exe file) as a dependency.

To add the module containing PrimeNumberChecker as a dependency to the application, we first get PrimeNumberCheckers's System.Type object and then use Type's Module property.

Having specified a) the Manager of the grid where our application is to be executed and b) the application's dependencies our application is ready to be executed. However, we also need a way to be notified when our threads have finished execution. For this, we subscribe to the ThreadFinish event of our GApplication, which is fired when a thread finishes execution.

```
...  
private static void App_ThreadFinish(GThread thread)  
{  
    // cast the supplied GThread back to PrimeNumberChecker  
    PrimeNumberChecker pnc = (PrimeNumberChecker) thread;  
}
```

```

// check whether the candidate is prime or not
bool prime = false;
if (pnc.Factors == 2) prime = true;

// display results
Console.WriteLine(
    "{0} is prime? {1} ({2} factors)", pnc.Candidate, prime,
pnc.Factors);
    }
}
}

```

The finished thread is provided as a parameter of the GThreadFinish event handler. The event handler casts the thread to a reference of the original type (PrimeNumberChecker), checks the number of factors to determine whether the number is prime or not and finally displays the result.

```

C:\WINDOWS\System32\cmd.exe - Pri...
47012 is prime? False (24 factors)
587512 is prime? False (32 factors)
913145 is prime? False (8 factors)
700014 is prime? False (24 factors)
327083 is prime? False (4 factors)
75189 is prime? False (8 factors)
748050 is prime? False (24 factors)
800224 is prime? False (24 factors)
909739 is prime? False (4 factors)
467339 is prime? False (6 factors)
910532 is prime? False (24 factors)
136949 is prime? True (2 factors)
903476 is prime? False (24 factors)
73994 is prime? False (4 factors)
430732 is prime? False (12 factors)
394320 is prime? False (80 factors)
23499 is prime? False (12 factors)
48812 is prime? False (6 factors)
649722 is prime? False (8 factors)
364940 is prime? False (24 factors)
656547 is prime? False (4 factors)
714338 is prime? False (4 factors)
207817 is prime? False (4 factors)
402207 is prime? False (8 factors)

```

Figure 13. Running the PrimeNumberGenerator grid application

Note: The source code for this tutorial can be found in the examples\PrimeNumberGenerator directory.

3.3. Other Features of the Alchemi API

Application ID

Once a GApplication is started via the Start method, an ID for it is created. This is accessible via the Id (string) property and looks something like 32bfb214-cec9-4277-816c-ac54637e8bea.

Thread ID

Similarly, GThread also has an Id (int) property. This value is automatically generated when a thread is added to an application. The first thread added will have an Id of 0, the second 1 and so on.

A specific thread is identified by its ID, which is used as an indexer in the Threads collection of GApplication to reference a particular thread. e.g.

```
GThread th = myApp.Threads[0];
```

Dependencies

The tutorial only uses the basic dependency (ModuleDependency) which specifies the .NET module containing the implementation of your GThread-derived class.

However, if the GThread uses additional DLL's that are not part of the .NET Framework, they need to be specified as dependencies as well. For example, the Mandelbrot application (\examples\Mandelbrot) uses complex.dll, which is a complex math library sourced from http://www.codeproject.com/dotnet/complex_math.asp.

Additionally, in the tutorial, all code is contained in one file and the application is compiled to an .EXE. Since only the implementation of the GThread needs to be shipped to an Executor, it can be separately compiled into a .DLL and specified as a module dependency. The benefit of this is a reduction in data transferred. The PiCalculator example (\examples\PiCalculator) uses this approach.

Finally, by using EmbeddedFileDependency, any file can be included as a dependency by specifying its location and the name that it should be recreated as. This could be used for example, to include a data file for the thread to process.

It should be noted that ModuleDependency is derived from EmbeddedFileDependency and simply provides a more convenient way of specifying a file's location and name.

Other Events

In addition to the ThreadFinish event GApplication has two other events that can be subscribed to. ThreadFailed is fired when a thread has failed. and ApplicationFinish is fired when all the threads in an application have finished executing.

Aborting a Thread

A thread can be aborted by calling its void Abort() method. e.g.

```
myApp.Threads[0].Abort();
```

Aborting an Application

An application can be aborted by calling its void Stop() method. This will abort all threads in the application. e.g.

```
myApp.Stop();
```

Thread State

A thread's state can be accessed via its State (ThreadState) property. e.g.

```
Console.WriteLine(myApp.Threads[0].State);
```

The ThreadState enumeration has the following values:

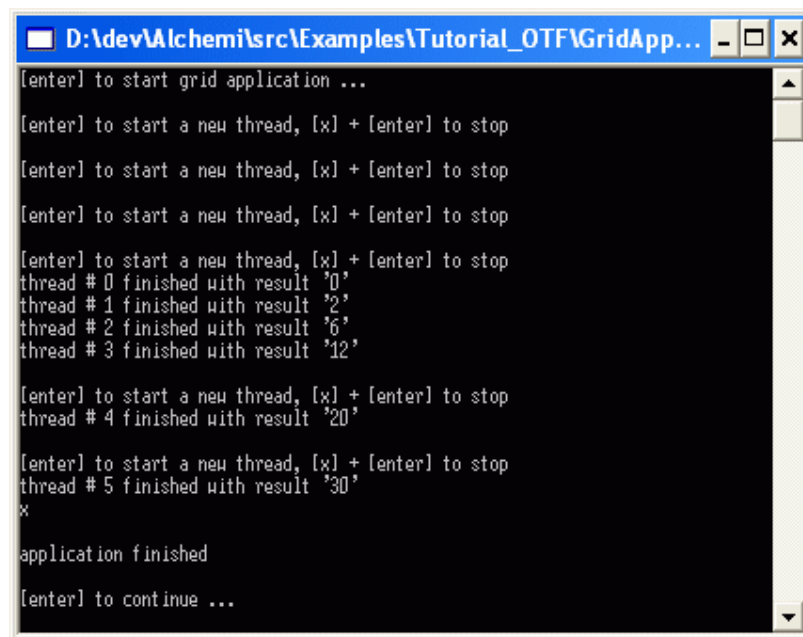
- Ready: The thread is ready to be executed
- Scheduled: The thread has been scheduled to run on an Executor and is being prepared for execution
- Started: The thread has started executing
- Finished: The thread has finished executing
- Dead: The thread has been aborted or it has finished executing (this includes having failed) and has been returned to the user

On-The-Fly Execution

In the tutorial, all threads are added to the application and it is then started. An optional scenario is on-the-fly execution. This is useful for real-time or "service" applications where information about all threads to be executed is not known beforehand. Please note the following for using on-the-fly execution.

- Call `GApplication.Start()` after all dependencies have been set
- Call `GApplication.StartThread(..)` to execute a thread on-the-fly
- You must call `GApplication.Stop()` once no more threads need to be executed. (This is not required if setting the `GApplication.ApplicationFinished` delegate, but setting this delegate usually does not make sense for on-the-fly execution.)

The directory `\examples\Tutorial_OTF` contains a version of the `\examples\Tutorial` application modified to demonstrate on-the-fly execution:



```
D:\dev\Alchemi\src\Examples\Tutorial_OTF\GridApp...
[enter] to start grid application ...
[enter] to start a new thread, [x] + [enter] to stop
[enter] to start a new thread, [x] + [enter] to stop
[enter] to start a new thread, [x] + [enter] to stop
[enter] to start a new thread, [x] + [enter] to stop
thread # 0 finished with result '0'
thread # 1 finished with result '2'
thread # 2 finished with result '6'
thread # 3 finished with result '12'
[enter] to start a new thread, [x] + [enter] to stop
thread # 4 finished with result '20'
[enter] to start a new thread, [x] + [enter] to stop
thread # 5 finished with result '30'
x
application finished
[enter] to continue ...
```

Figure 14. Example of on-the-fly execution

Synchronisation & Workflow

Although grid threads cannot communicate with each other, logic that actions on a thread based on the completion of another - e.g. the result of a particular thread is needed for a subsequent thread - can be incorporated into the `ThreadFinish` event handler.

Thread Priorities

You can request a priority by setting the `Priority` property for a thread before it is executed. The highest priority is 0 with larger integers denoted lower priorities. The default priority is 5.

3.4. Grid Job Model : Grid-Enabling Legacy Applications

Alchemi supports the traditional job model i.e. jobs within tasks, where each job is a process with input and output files.

It should be noted that in Alchemi:

- a task is just another name for a grid application and
- internally, a job is an instance of the `GJob` class which inherits from the `GThread` class (i.e. a job is actually just a specialised grid thread)

This means that any tools or code that work with grid applications and grid threads will also work with tasks and jobs respectively.

There are two options available for grid-enabling existing applications:

- Programmatically via the API
- Declaratively via the Alchemi Job Submitter

Programmatically Composing and Executing Jobs Using the Alchemi API

The JobAPI example ([SDK]\Examples\JobAPI example demonstrates the use of the Alchemi API for tasks/jobs. The solution contains two projects: Reverse (a simple console application that reverses the text of a file specified as a command-line argument and displays the results) and GridReverser (a console application that demonstrates the grid-enabling of Reverse).

The Alchemi Job Submitter

The Job Submitter is a console application ([SDK]\alchemi_jsub.exe) can be used to submit tasks/jobs and retrieve their results.

Its use is demonstrated here by a simple example. The files required for this example can be found in [SDK]\Examples\JSubDemo.

The file reverse.task.xml contains an example representation of a task:

```
<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>
```

Example task representation.

The Job Submitter must be started with the host and port of an Alchemi Manager and user credentials. The following screenshot shows how the Job Submitter can be used to submit a task, monitor its jobs and retrieve results:

```
C:\WINDOWS\System32\cmd.exe

C:\akshay\dev\Alchemi\project\examples\JSubDemo>..\..\bin\SDK\alchemi_jsub localhost 9000 user user

Alchemi [.NET Grid Computing Framework]
http://www.alchemi.net

Job Submitter v0.8.1
Connected to Manager at localhost:9000

> submittask reverse.task.xml
-- Task submitted (alias = 1010).

> getfinishedjobs 1010
-- Got 2 job(s).
-- Wrote file .\result1.txt for job 0.
-- Wrote file .\result2.txt for job 1.

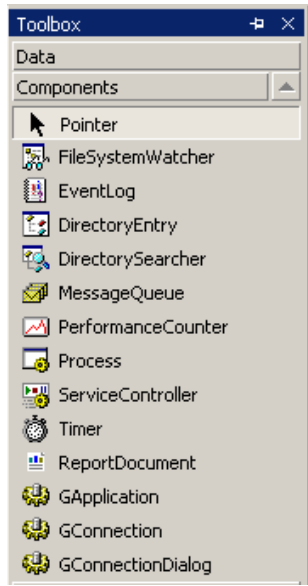
> q

C:\akshay\dev\Alchemi\project\examples\JSubDemo>_
```

There are other commands you can use as well; invoke alchemi_jsub.exe without any arguments for help.

3.5. Drag & Drop Components

Alchemi provides a simple programming environment, including drag-and-drop components for building grid-applications using Visual Studio .Net.

<p>The following via drag-and-drop components are available:</p> <ul style="list-style-type: none">• GApplication• GConnection• GConnectionDialog	 <p>Figure 15. Alchemi Drag-and-drop components</p>
---	--

The GApplication, GConnection can be used as described in the tutorial section. The GConnectionDialog is a standard login dialog with host and port which can be used to authenticate to the Manager.

3.6. Cross-Platform Development: The Cross-Platform Manager Web Service

The Alchemi Manager provides a web-service which can be used to submit legacy applications to an Alchemi grid. This web service is known as the Cross-platform Manager. The application is described using the Alchemi's own xml format, (described in section 3.4 Grid enabling legacy applications).

In the current version, the CrossPlatform Manager provides the functions shown in figure 15. The webservice can be used to submit jobs to Alchemi from remote computers, using languages that are not .Net compatible, for example Java. We also plan to provide a Java API to build Alchemi clients and enable easier job-submission, querying and monitoring interfaces, in the near future.

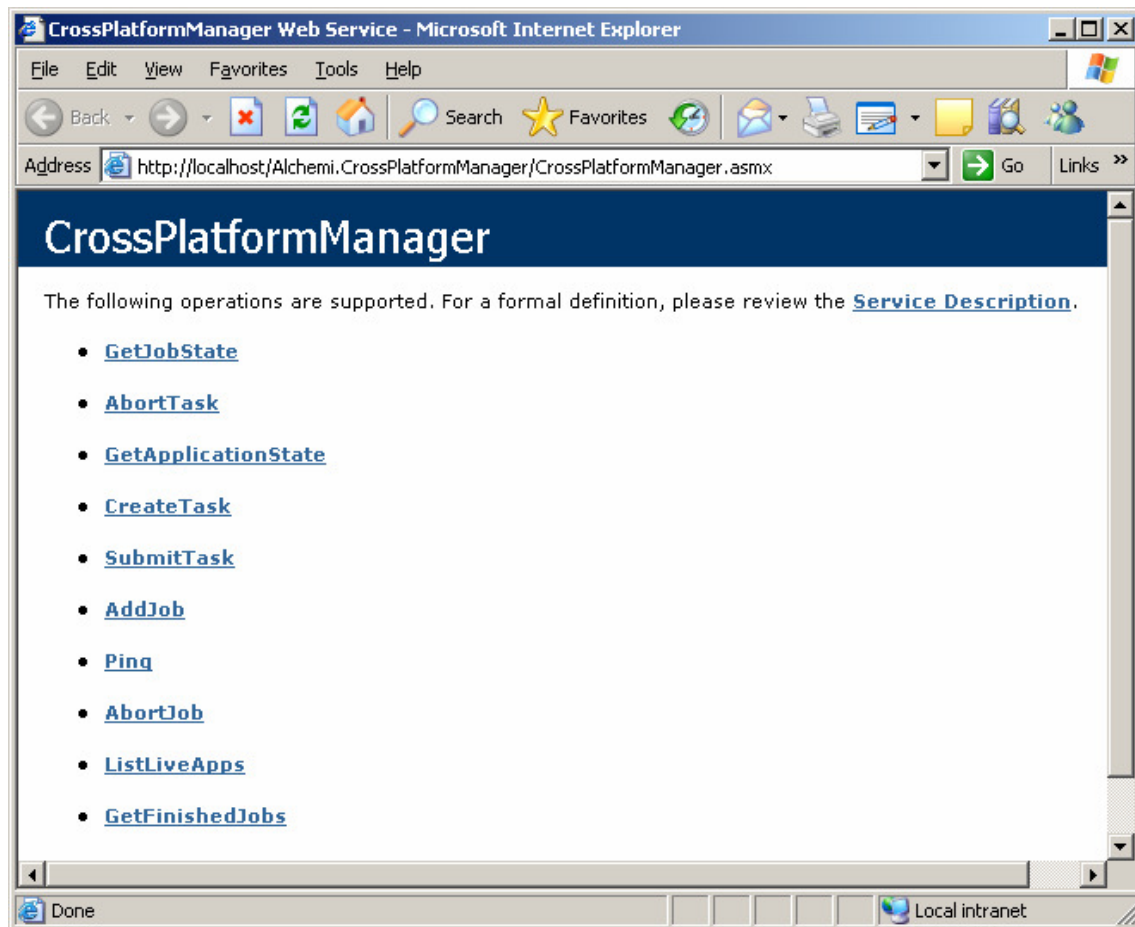


Figure 16. Alchemi CrossPlatform Manager

4. Conclusion

Alchemi is an easy-to-use .Net Grid-computing framework that aims to lower the barrier of entry into the world of distributed / Grid computing, by making it easier to setup, manage and maintain Grids on a local or wide-area network. The framework is in constant development, and as an open-source project under the GPL, we encourage community participation and feedback in all aspects of the development of Alchemi.

More information will be added to this manual as additional features are available in Alchemi. Please direct any feedback about Alchemi and/or this manual to Krishna Nadiminti (kna@cs.mu.oz.au).

We would like to thank the entire Gridbus team for their comments, support and contribution to the development of Alchemi.

5. Alchemi References

- [1] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan & Srikumar Venugopal, [Alchemi: A .NET-based Grid Computing Framework and its Integration into Global Grids](#), Technical Report, GRIDS-TR-2003-8, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, December 2003.
- [2] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal, [Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework](#), *High Performance Computing: Paradigm and Infrastructure*, Laurence Yang and Minyi Guo (editors), ISBN: 0-471-65471-X, Wiley Press, New Jersey, USA, June 2005.
- [3] Agus Setiawan, David Adiutama, Julius Liman, Akshay Luther and Rajkumar Buyya, [GridCrypt: High Performance Symmetric Key using Enterprise Grids](#), Proceedings of the 5th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2004, December 8-10, 2004, Singapore), Springer Verlag Publications (LNCS Series), Berlin, Germany.
- [4] Krishna Nadiminti, Yi-Feng Chiu, Nick Teoh, Akshay Luther, Srikumar Venugopal, and Rajkumar Buyya, [ExcelGrid: A .NET Plug-in for Outsourcing Excel Spreadsheet Workload to Enterprise and Global Grids](#), Proceedings of the 12th International Conference on Advanced Computing and Communication (ADCOM 2004, December 15-18, 2004), Ahmedabad, India.
- [5] Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar 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.