EPCC-SS-2000-02

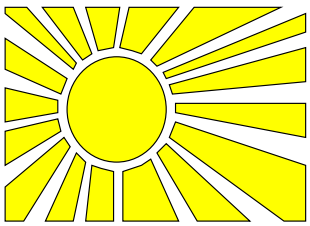
JiniGrid: Specification and Implementation of a Task Farm Service for Jini

Benjamin Clifford

September, 2000

Abstract

Jini is a technology that allows networks of heterogeneous services to organise themselves with little human intervention. It provides for fault tolerance and for the automatic transfer of driver code, written in Java, when and where it is needed.
These abilities will be extremely useful, if not essential, in a large scale environment such as the Grid.
In this project, Jini technology is used to share compute servers using the task farm paradigm.
A standard API that all compute servers must implement is defined. Four implementations with different capabilities are presented as well as a selection of clients and helper classes.



Contents

1 Introduction

This project aims to use Jini and Java to build a small prototype Grid.

The Grid is a concept that will provide the ability for a user to use some high performance computing resource without knowing exactly where or what it is. One part of The Grid will be the use of remote computation servers, selected either by the user or by some automatic process, based for example on the amount of free processor time or on cost.

Java provides cross-platform compatibility between a wide variety of platforms, by providing abstractions of common system facilities such as the file-system, as well as techniques for transferring code and data between disparate systems.

Jini, building on top of the Java platform, provides the ability for clients and services to discover and communicate with each other with a minimum of knowledge about each other.

Task farms are a good abstraction for this as they are quite independent of the underlying parallel architecture. Research has already taken place within EPCC on task farms in Java, and it is upon a previous EPCC-SSP report that I have based my work.[4]

In this project, I specify a standard service interface for task farms in Jini and four implementations of this interface - compute servers with a range of abilities. I also present several example clients, ranging from a simple integral computation to a more advanced graphical interactive program and a number of utility scripts and classes.

This project was targeted at developing prototype systems. Further work could be done to improve efficiency and error handling.

Many questions have been raised during the development of this project, which I have noted either in the relevant sections or at the end of this document.

2 Background

2.1 About the previous HPT project

The project of two years ago implemented HPT (Hitachi Parallel Tasks), a remotely accessible task farm system, intended to run Java code primarily on the Hitachi SR2201 using MPI but also capable of running on other MPI systems and on shared memory systems.

One of the ideas behind the design was that a client should be able to run with minimal code modification on a workstation for initial development, on larger systems for testing and finally on the Hitachi machine for real-world usage.

The project did not go so far as to provide a completely generic API. In order to run a task farm on the local machine, two different forms of invocation were necessary, depending on whether the user wished to use shared memory or MPI for communication. In order to use a remote compute server, a third form of invocation was required. Furthermore, a single-threaded task farm suitable for use on a uniprocessor machine was not provided - the user was expected to write his own or to put up with the (sometimes large) overheads of running the multiprocessor code.[1]

The project did not provide a mechanism for clients to locate compute servers on the network. It was necessary to know on which machine the server was running and pass the hostname to the HPT code.

Most of HPT has been used in this project. Some classes were used as part of the specification, whilst the main server code became ServerOne.

2.2 About the Task-farm paradigm

A task farm consists of a number of worker processes, coordinated by a master process. Typically each worker process runs on its own processor. The master may have its own processor or may use time on the worker processors.

The calculation to be performed is broken down into a number of component calculations (called tasks) which can each be evaluated independently.

All of the tasks are distributed to the workers, which evaluate them and return the results to the master process. The master process combines all of the individual results into an overall result.

Because each task is independent, it can be computed on any processor. The master can therefore dynamically distribute tasks as processors become free, minimising processor idle time.

[1] The idle loop of the HPT master thread does not release processor time whilst it is idling, so running with a single worker means that only half of the processor time can be used for actual task computation.

2.3 About Jini

Jini is a collection of classes and concepts that enables clients and services running on a network to discover each other and provides standard protocols for the clients and services to interact once they have done so. It makes heavy use of Java features such as object serialization, remote method invocation (RMI) and bytecode portability.

Ideally, new clients and services can be added to the network and Jini will allow them to communicate without the need for the use to install new drivers. A server can provide code for execution on the client to facilitate communication.

A service can be a provided by a physical device (for example, a printer or disk drive), or can be more abstract (such as a directory service). A service is distinct from a server, which is the physical platform on which the service runs.

The Jini specification provides a number of standard services. Of fundamental importance is the lookup service. Services register service objects with the lookup service when they are started. Clients may interrogate the lookup service to discover relevant services.

Attributes can be associated with each service object. A client may specify a template of attributes when searching for services, to restrict the selection of services returned.

Inter-operation between different implementations of the same type of service, possibly provided by different authors, is achieved by clearly separating the definition of the *service interface* from the implementations. Clients and servers are written to this interface definition and are therefore (hopefully) inter-operable.

The service object, which is transferred to the client machine, generally needs to communicate service requests back to its own server. It may do this in any way. Commonly RMI is used as very little coding is required. However, other protocols, either standard or custom, may be used.

One service object may implement several service interfaces. For example, many Jini services also implement the Administrable interface, which permits administration of the service in a standardised manner.

Jini clients and services should be fault tolerant. A service must assume that a client could disappear (for example, crash or be cut off from the network) at any time, and vice versa.

Services should be capable of releasing resources allocated to such a client automatically. This is usually achieved with the concept of leasing: a client must regularly tell the server that it is still alive in order to maintain its resource allocations. If it does not, its lease expires and the resources are freed. For example, a service must lease its registration with the lookup service to prevent the lookup service filling up with crashed services and to prevent the supply of long dead services to clients.

Several lookup services may be run on the network. Services should register with each lookup service that they find, and clients should interrogate all lookup services when searching for a particular service.

Lookup services belong to one or more *discovery groups*. By default, services are in the *public group*.

Clients may restrict their lookups to services registered in particular groups. For example, a departmental printer could expose itself only to lookup services within the departmental group.

Discovery groups are not intended to hide devices for security purposes, but rather to reduce the load on the network by not providing clients with services that they cannot use.

2.4 About the systems used at EPCC

Development and testing occurred on the following systems:

Amber and Beryl: single processor Sun systems. Amber has one UltraSPARC II 300MHz processor, whilst beryl has an UltraSPARC 167MHz processor.

BOBCAT: BOBCAT is a Beowulf machine consisting of a master PC and 16 worker PCs. Each worker has a 650MHz Athlon processor, with 128Mb of RAM. The operating system used is Red Hat GNU-Linux. There is no formal queueing system; many jobs may run simultaneously.

The Lomond service: The Lomond service consists of three Sun shared memory machines:

Name	Processors	Speed (MHz)	Total Memory (Gb)
E3000	4 x UltraSPARC	250	1
HPC3500	8 x UltraSPARC II	400	8
HPC6500	18 x UltraSPARC II	400	18

These systems cannot be used interactively for executing substantial pieces of code; such jobs must be submitted through a queueing system, LSF, using the bsub command[1]. All of the queues are managed by the E3000.

3 The JiniGrid API

3.1 Introduction to the API

The API consist of three interfaces which specify the API provided by compute servers, two Entry classes specifying attributes of the compute servers, a class which encapsulates a Jar file and an Event.

It is contained in the `jinigrid.spec` package, which is analogous to the `net.jini.core` packages. This package must be present on every JiniGrid system.

The JiniGrid API is based substantially on the HPT API, although I have attempted to follow the Jini style, where abstraction is important. As it stands, further abstraction is possible, for example by replacing JarData with some form of Map.

Clients and services must set a security manager with the `System.setSecurityManager` method, as RMI will not download code unless one has been installed. Very few permissions need to be granted to foreign code. An example security policy used during testing is supplied in the file `standard.policy`.

The registration of a `TaskFarmService` with a lookup service includes the specification of entries containing attributes. A `TaskFarmService` should provide a `net.jini.lookup-.entry.Name` entry with the human readable name of the service. Additionally, services

may specify a `TaskFarmInfo` object and as many `SupportedAPIEntry` objects as is necessary.

The client must provide a class implementing the `Slaveable` interface. An instance will be created on each worker processor. Client specified global data will be passed to each `Slaveable`, then the task farm will begin passing in task objects. There are no constraints on the class of the global and individual task objects - depending on the circumstances, it might be desirable to pass a user-defined structure or merely a wrapper class such as `java.lang.Integer`.

The client must provide a Jar containing all non-system classes used by the objects sent between client and server.

The task, global data and result objects must all be serializable.

A client may register to receive `ProgressEvent`s that indicate how far the task farm has got in its processing.

3.2 TaskFarmService

This is the base interface through which everything happens.

Services register a `TaskFarmService` object with the lookup service. When `TaskFarmService.getTaskFarm()` is called, an object implementing the `TaskFarm` interface is produced. This may involve setting up a communication session with the compute server.

```
public interface TaskFarmService
{
   public TaskFarm getTaskFarm(String username,
                               String keyfilename,
                               int numSlaves, JarData jarFile)
                               throws RemoteException;
}
```

3.3 TaskFarm

A `TaskFarm` object will execute a specified vector of tasks, returning a vector of results.

Objects implementing this interface are usually acquired by a user program by using the `getTaskFarm` method of a `TaskFarmService` object, although this is not always the case (for example, the ServerThread implementation can be used independently).

The `TaskFarm` is used by calling the `runTask` method, passing in three parameters - a `Vector` of tasks (which may be of any class), a global data `Object` which will be passed to each slave and a `String` containing the name of a `Slaveable` class. Instances of this class will be created to perform the tasks.

When the `TaskFarm` is finished with, the `finish()` method should be called.

Once a `TaskFarm` has been acquired, it may be used as many times as desired.[2]

[2] This is the reason that the Jar file is specified in the task farm factory method, not in the `runTask` method as is the case with HPT - it is inefficient to send the same Jar repeatedly again.

A `TaskFarm` may only be used by one thread at once, although a `TaskFarmService` may be used by multiple threads to generate multiple `TaskFarm` objects, one for each thread; and once a run has completed, a `TaskFarm` can be used by another thread.

```
public interface TaskFarm
{
   public Vector runTask(Vector tasks,
           Object globalData,
           String slaveClass)
           throws RemoteException;
   public void finish()
           throws RemoteException;
   public EventRegistration trackProgress(long duration,
           RemoteEventListener rel,
           MarshalledObject key,
           int reportEvery,
           boolean sendResults)
           throws RemoteException;
}
```

3.4 Slaveable

Implementations of this interface are used to evaluate tasks. Classes that implement this must provide a no-argument constructor.

Method calls will be made in the order presented below. All except the first may be repeated multiple times in the lifetime of the `Slaveable` object.

```
public interface Slaveable extends java.lang.Runnable {
   public void setInitAllTasks(Object globalData);
   public void setTask(Object task);
   public void run();
   public Object getResult();
}
```

3.5 TaskFarmInfo

This is an `Entry` class that provides general information about a task farm's services.

numProcessors provides the maximum number of processors that a `TaskFarm` from the service may use.

infoURL provides a URL to a location where human readable information may be found about the service.

architecture indicates the communications architecture used by the server. It must be set to one of the static constants provided in this class.

```
public class TaskFarmInfo extends net.jini.entry.AbstractEntry
{
   public Integer numProcessors;
   public java.net.URL infoURL;
   public Object architecture;

   static final public Object ARCH_MSG;
   static final public Object ARCH_SHMEM;
   static final public Object ARCH_SINGLE;
}
```

3.6 SupportedAPIEntry

This `Entry` specifies an API that is supported by a JiniGrid server. Instances provide a string name of the API and a multi-part version number

Each server should have at least one `SupportedAPIEntry`, specifying the API name as "java" and the version of Java supported. Such an entry may be fabricated using the `static getJavaEntry()` method.

```
public class SupportedAPIEntry
         extends net.jini.entry.AbstractEntry
{
   public SupportedAPIEntry();
   public SupportedAPIEntry(String apiName,
                            MultiVersion apiVersion);
   public SupportedAPIEntry(String apiName,
                            String apiVersion);

   public String name;
   public MultiVersion version;

   public static SupportedAPIEntry getJavaEntry();
}
```

3.7 JarData

`JarData` encapsulates a previously created .jar file into a serializable object that may be transmitted across the network. The filename containing the data is specified in the object constructor.

This class was taken from HPT, although most of the methods have been removed for the sake of simplicity.

```
public class JarData implements Serializable
{
  public JarData(String fn) throws IOException;
  public byte[] getJarData();
}
```

3.8 ProgressEvent

A `ProgressEvent` is fired when a certain number of tasks have been completed by a `TaskFarm`. It may contain the results of those tasks or may merely notify the client that a certain number of tasks have been completed.

The `TaskFarm.trackProgess` method should be called by a client to indicate that it wishes to receive progress events:

```
public EventRegistration trackProgress(long duration,
         RemoteEventListener rel, MarshalledObject key,
         int reportEvery, boolean sendResults)
         throws RemoteException;
```

duration specifies the desired lease duration for the event registration, in milliseconds.

rel specifies the `RemoteEventListener` that will receive the events.

key is a cookie which is sent back with every `ProgessEvent`. The client may use this to keep track of where the event is coming from.

reportEvery specifies the number of tasks that should be completed between each ProgressEvent.

sendResults specifies whether results should be sent back as part of the `ProgressEvent`. `true` means that the results should be sent back. If this is the case, the `newResults` member of the `ProgressEvent` will contain a `Vector` of these results.

An instance of this class contains the number of tasks completed, and a `Vector` of new results. If no results are being supplied, then either the `newResults` variable should be null or should point to an empty `Vector`.

The number of results in the vector is not necessarily equal to the value in `cumulativeTasksCom-pleted`, and neither of these are necessarily equal to the number of results requested in the trackProgress method call.

A constant, `PARTIAL`, is defined as the only event ID.

```
public class ProgressEvent extends RemoteEvent
{
   public final int cumulativeTasksCompleted;
   public final Vector newResults;

   public ProgressEvent(Object source,
             long eventID, long seqNum,
             MarshalledObject handback, int ctc,
```

```
            Vector nr)

   static public final long PARTIAL;
}
```

4 Environment and Operation

4.1 Software versions used

- Jini:
  `Jini 1.1beta`
- JDK on the Sun machines:
  `java version "1.2.1"`
  `Solaris VM (build Solaris_JDK_1.2.1_02, native threads, sunwjit)`
  There appears to be a bug associated with Java whereby processes would lock up. This appeared in the lookup service on both beryl and amber: after around 30 seconds of CPU time have been used, the lookup service will occasionally hang. The OGR client also repeatedly hangs after around 10 minutes of computation.
- JVM on BOBCAT
  `java version "1.2"`
  `Classic VM (build Linux_JDK_1.2_pre-release-v2, native threads, sunwjit)`

4.2 The Lookup Path

Jini provides for the discovery of lookup services through several mechanisms. When using *multicast discovery*, Jini does not need to know the location of the lookup service - it can find any lookup service within the multicast radius.[3] *Unicast discovery*, on the other hand, requires precise knowledge of the location of the lookup service (specified by a URL of the form `jini://hostname/`) but works over the whole Internet.

Multicast routing is usually not configured on most networks, so to discover services over a wide area, it is necessary to use unicast discovery.

Jini provides for unicast *lookup locators*, pointing at a program specified host, but does not define any standard way for a program to be configured to use one.

I define, therefore, the concept of a lookup path. This is similar in syntax to the unix command path, and specifies a list of discovery groups and lookup locators to be used for service location.

The lookup path takes the format:

`<specifier>[,<specifier>[,<specifier>[...]]]`

where `<specifier>` may be either a lookup discovery group or a `jini://` URL.

[3] Unless multicast routers are present, the multicast radius encompasses only the local IP network. In EPCC, there are several IP networks with no multicast routing between them, so multicast discovery generally does not work.

In development, the following path was used:

```
EPCCGROUP,jini://amber.epcc.ed.ac.uk/,jini://beryl.epcc.ed.ac.uk
```

This means that clients and servers would make a multicast broadcast to lookup services in the group EPCCGROUP, whilst also attempting to explicitly contact lookup services running on amber and beryl.

An alternative to using a lookup path would be to ensure that a lookup service and lookup discovery service[4] are running in each multicast-connected region, with the lookup discovery services correctly configured to be fully connected to each other.

In an environment where Jini is ubiquitous, this would require less configuration. That scenario is a long way off and may never happen. However, this model would still help on a network where several systems are making use of remote compute services - the paths to the remote systems need only be configured once per network, in the lookup service, rather than each client needing to configure a lookup path.

These two systems are not mutually exclusive. A client can use both a lookup path and lookup discovery services to locate services.

4.3 Environment variables

These environment variables are used by many of the supplied scripts to set the corresponding system properties.

$JINIGRID_POLICY The path to the Java security policy file to be used by JiniGrid clients.

$JINIGRID_LOOKUP The lookup path

4.4 System properties

serverOne.mode specifies the mode that ServerOne should run in:
`mpi` means MPI mode
`pipes` means shared memory mode.

serverOne.numslaves specifies the number of slaves to be used in shared memory mode. In MPI mode, this property is unused as the number of processors can be obtained from the MPI library.

serverOne.userfile specifies the filename of the public key file for user authentication.

jinigrid.packageversion specifies the filename of the package version specification file for constructing `SupportedAPIEntry` attributes. See section 4.5.

jinigrid.lookup specifies the lookup path.

jinigrid.serverQueue.JobID is used internally by ServerQueue to communicate between the queue manager and workers.

[4] Lookup Discovery services provide a bridge between lookup services.

java.rmi.server.codebase specifies the URL(s) of RMI stubs. This property should point to a URL where the JiniGrid utility stubs can be loaded, if any of the utility classes are used. Additionally, if the client/server exports any other stubs, such as for remote events, then the URLs for these should also be present.
This property is discussed in more detail in [3].

4.5 Files

.jar files The various jar files that are part of this project are listed in table 1.

jinigrid-spec.jar	Contains the JiniGrid API classes. The contents must be present on every JiniGrid system, client or server.
jinigrid-util.jar	The utility classes, that are not strictly necessary for JiniGrid compatibility, but are very useful.
serverone.jar	The classes necessary to run ServerOne.
serverthread.jar	The classes necessary to run ServerThread.
serverqueue.jar	The classes necessary to run ServerQueue.
servermeta.jar	The classes necessary to run ServerMeta.

Table 1: .jar files and their contents

package.versions is a sample package version information file used to create `SupportedAPIEntry` objects. The format is in Java property file format, with one package per line, looking like:
`MagicMathsLib 1.3`

standard.policy specifies a simple security policy. Locally loaded code is given full permissions, whilst code loaded over the network is severely restricted. The permissions consist of ability to connect to any system on the network, to read any property with a name beginning with `jinigrid.` and to read a specific file containing the user's private key.

The latter is included because the ServerOne code needs this for authentication. See section 12 for notes on user authentication.

gridclient is a script for running clients with the environment correctly set. See section 11.1.

start.serverOne launches serverOne in threads mode, for use on the various Sun systems.

start.bobcat is a script to start serverOne on BOBCAT, in MPI mode. It is a modification of the prunjava script that is part of mpiJava.

start.serverThread launches serverThread.

start.serverQueue launches serverQueue.

start.serverMeta launches serverMeta.

4.6 Compiling the code

All compilation of code took place on amber and beryl. As far as I am aware, there is nothing specific to the version of the JDK and the code should be compilable on any version of Java >=

1.2.

There are Makefiles in most directories – these mostly call standard commands: javac, rmic, javadoc.

To compile serverOne, it is necessary to have the mpiJava .class files, but it is not necessary to have a working installation of mpiJava.

The documentation section of the master Makefile uses a latex doclet, available at the time of writing from `http://www.c2-tech.com/java/TexDoclet/`

The entire project can be built by simply typing
`make`
whilst in the top level directory.

4.7 Directory Structure

All of the code and support files for the project are held in a CVS module, `ssp-2000-02`.

In the top level directory of this module are found all of the non-code files - the scripts and example configuration files.

The `jinigrid/` subdirectory contains the source code. As is usual, code in different packages appears in subdirectories within this – for example, the source for `jinigrid.util.ServiceFinder` can be found in `jinigrid/util/ServiceFinder.java`.

4.8 An Example JiniGrid - instructions

Pre-requisites: Jini, MPI, mpiJava.

1. Compile the code (see previous section).
2. On some host[5], run `rmid` and the Jini lookup service.
3. On some host, run a JiniGrid service, for example, ServerThread. Instructions on how to do this for each service are supplied in the following sections.
4. On some host, run a client, for example, fractalBrowse. Instructions are supplied in the relevant section.

5 ServerOne - An HPT based JiniGrid server

5.1 Structure of HPT

HPT was implemented in [4]. It provides a compute server program and client side code (the `TaskRunner` class) to access that compute server.

[5] Because of Jini's distributed nature, it does not matter particularly on which hosts this and the following steps are performed - one host could be used for all steps, or different hosts could be used for each.

Communication, both between client and server and within the components of the server, takes place with message passing. Each type of message is represented by a subclass of `Message`. All messages contain a payload object, inherited from `Message`. Subclasses can contain additional data. The various message types are detailed in table 2.

Message passing within the server, between the master component and worker components, is encapsulated within a communication wrapper class, a subclass of `CommsWrapper00`. There are two subclasses, corresponding to MPI and shared memory architectures.

Table 2: Message types used in HPT

Class Name	Direction	Comment
Message	-	Superclass of all messages
CMadminMsg	-	Superclass of administration messages
CMadminDestroyMsg	Client to Server	Causes the server to shut down.
CMdieMsg	Client to Server	Tells the server that the client is finished
CMinitMsg	Client to Server	Initialises the server
CMinitReplyMsg	Server to Client	Returns the result of initialisation.
CMjarMsg	Client to Server	Carries a JarData.
CMpartialResultMsg	Server to Client	Sent every time a task is completed.
CMresultMsg	Server to Client	Sent at the end of execution
CMrunSlaveMsg	Client to Server	Causes the server to commence execution
MSclassNameMsg	Master to Slave	Conveys the name of the Slaveable.
MScontrolTaskMsg	Master to Slave	Controls how a task is run.
MSinitAllTasksMsg	Master to Slave	Conveys the global data.
MSkillSlaveMsg	Master to Slave	Causes the slave to die
MSresultMsg	Slave to Master	Conveys a completed result to the master.
MSsetGroupMsg	Master to Slave	Used for setting MPI process group
MStaskMsg	Master to Slave	Conveys a task to be performed.
MSunsetGroupMsg	Master to Slave	Tells a slave to be in no process group.

HPT has a number of inefficiencies, the most noticeable being that it uses a large amount of CPU time whilst idling, on both the client and server sides.

5.2 Changes made to HPT

Very little of HPT needed to be modified to adapt it for use with the JiniGrid API. Mostly the code consists of a wrapper around the `TaskRunner` class. It was necessary to modify the code as follows:

Where the HPT code originally established a `ServerSocket` to accept connections, code was added that also registered a service object with the Jini lookup service. This object carries the `TaskRunner` code with it and wraps JiniGrid calls into `TaskRunner` calls at the client.

To implement administration, it was necessary to create the admin. message types.

To implement progress events, it was necessary to create `CMpartialResultMsg`.

Independent of the effort to port HPT to JiniGrid, the following changes were made:

Some rearrangement of the HPT classes was made in order that a single build would support

both MPI and shared memory. Two separate classes both called `CommsWrapper` (one for MPI and one for threads) were renamed to `MPICommsWrapper` and `PipeCommsWrapper`, respectively.

The `MPICommsWrapper` class was modified to use new a Java/MPI binding, mpiJava. Previously the code used JavaMPI which is now obsolete. [cite mpiJava reference]

The handling of jar files was changed so that on-disk storage is no longer used.

5.3 Configuration and Use

The server may be started using the `start.serverOne` script on one of the Sun systems, or `start.bobcat` on BOBCAT.

5.3.1 Installation and Use on BOBCAT

mpiJava must be installed. It is assumed to be installed in `$HOME/mpiJava`. If it is installed elsewhere, modifications should be made to the following lines.

These lines are taken from my `.profile` on BOBCAT:

```
PATH=$HOME/mpiJava/src/scripts:$PATH
CLASSPATH=$HOME/mpiJava/lib/classes/:$CLASSPATH
CLASSPATH=$HOME/libs/jinigrid-util.jar:$CLASSPATH
CLASSPATH=$HOME/libs/jinigrid-spec:$CLASSPATH
CLASSPATH=$HOME/libs/jini-core.jar:$CLASSPATH
CLASSPATH=$HOME/libs/sun-util.jar:$CLASSPATH
CLASSPATH=$HOME/libs/jini-ext.jar:$CLASSPATH
CLASSPATH=$HOME/libs/serverone.jar:$CLASSPATH
CLASSPATH=$HOME/libs/jinigrid-spec.jar:$CLASSPATH
export CLASSPATH
LD_LIBRARY_PATH=$HOME/mpiJava/lib:$LD_LIBRARY_PATH
LD_LIBRARY_PATH=/usr/local/mpich/lib/shared:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
export JINIGRID_POLICY=$HOME/standard.policy
export JINIGRID_LOOKUP=EPCCGROUP
```

It is necessary to have the following files installed in `$HOME`: `start.bobcat`, `standard.policy`, `package.versions`, and `userfn` (which is the key file for authentication).

In `$HOME/libs`, there should be `jinigrid-util.jar`, `jinigrid-spec.jar` and `serverone.jar` from JiniGrid, and `sun-util`, `jini-ext.jar` and `jini-core.jar` from Jini.

With the above in place, the server can be started with:

```
./start.bobcat
```

6 ServerThread - An RMI based JiniGrid server

ServerThread is a second implementation of the JiniGrid API. It uses RMI for client/server communication.

HPT uses its own protocol, because it appears that RMI was regarded as unreliable at the time of the previous project. However, testing RMI using this fresh implementation revealed no problems (in fact, fewer problems than were found with HPT).

The service object is an RMI stub, so calls are transmitted automatically to the server where they are executed.

ServerThread is designed for single-processor and shared-memory multiprocessor systems. The master and workers run as threads in a single JVM. Tasks and results are passed back and forth via a shared `Vector` object.

The ServerThread task farm class, `ThreadedTaskFarm`, may be instantiated by a client without using Jini, if it requires a local task farm. (This is used, for example, to provide the "My Computer" option for `TaskFarmSelectorJMenu`, in section 10.3).

6.1 Structure

The service object, `jinigrid.serverThread.Service`, is a `Remote` object that will manufacture a new instance of `jinigrid.serverThread.ThreadedTaskFarm` on demand.

An instance of `ThreadedTaskFarm` contains two main vectors, `tasks` and `results`. These form the basis for interprocess communication.

When a job is started, `ThreadedTaskFarm` creates the requisite number of worker threads, using itself as `Runnable` object. Each thread, therefore, shares access to the same task and result vector.

Each thread removes a task from the tasks vector, performs it, and places the result in the result vector. A thread terminates when it cannot find any more tasks to run.

The master launches the threads and then joins with all of them. When all of the threads have terminated, all of the results have been stored, and the `runTask` method can return.

Events are handled using the `ProgressEventHandler` class. Requesting a new event registration with the `TaskFarm.trackProgress` method causes a new `ProgressEventHandler` to be instantiated. This is not a `Remote` object and remains on the server.

The `ProgressEventHandler` receives notification from the `ThreadedTaskFarm` whenever a result is posted, and is responsible for counting the number of results so far to determine when a `ProgressEvent` should be sent. `ProgressEventHandler` also handles the leasing of the event registration.

6.2 Configuration and Use

The `CLASSPATH` environment variable should point to the JiniGrid specification and utility class files and to the ServerThread class files. It should also point at the Jini files.

An example is:

```
CLASSPATH=.:~clifford/jini1_1/lib/jini-core.jar:
           ~clifford/jini1_1/lib/jini-ext.jar:
           ~clifford/jini1_1/lib/sun-util.jar:
           ~clifford/ssp-2000-02/jinigrid-spec.jar:
           ~clifford/ssp-2000-02/jinigrid-util.jar:
           ~clifford/ssp-2000-02/serverthread.jar
```

The `package.versions` and `standard.policy` files must be installed, as well as `start-.serverThread`. It may be necessary to modify the latter if paths are changed.

The server can be started by typing:

```
./start.serverThread
```

7 ServerQueue - A Queueing server

On the Sun service machines, large jobs cannot be executed directly. Instead, they must be submitted to a queueing system and wait their turn for the processors. ServerQueue provides a wrapper around this queueing system so that clients may use the machines through the `TaskFarmService` interface, without needing any special code.

ServerQueue requires `rmiregistry` to be running in order for the queue manager, which must run on the E3000, to communicate with the worker processes which may run on any of the three component machines.

The code does not handle any task farm processing itself. Instead, it spawns an instance of ServerThread on the worker machine to process the job.

Whilst the program is hard-coded to use the `bsub` command and ServerThread, these platform-specifics are in a very small number of clearly identifiable places, so could be easily changed to other task farms or queueing systems.

7.1 Structure

The code is split into two main parts, the service and queue management side and the worker side. The service side receives jobs from clients, stores their details in a `ReliableMap` and submits a worker job to the queueing system using the `bsub` command.

When the worker is eventually run, it connects to the queue manager using RMI, retrieves the job and passes these to an instance of ServerThread. The results are then passed back to the queue manager and from there back to the client.

A single server may register several service objects, one for each supported queue. Only one instance of the queue manager is needed to manage them all.

7.2 Configuration and Use

Usage is almost identical to that of serverThread (section 6.2). It is necessary for the class path to point to `serverqueue.jar` in addition to `serverthread.jar`.

The server can be started by typing:

```
./start.serverQueue
```

8 ServerMeta - a meta-server

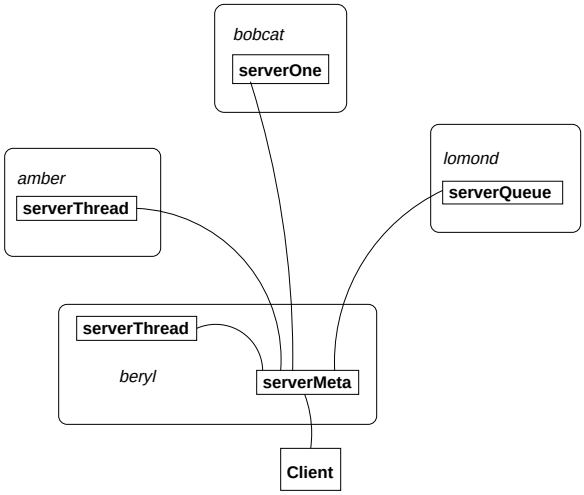


Figure 1: Metaserver combining several EPCC resources for a single client

ServerMeta is a "meta-server." Tasks submitted to a ServerMeta service are automatically broken up and distributed to the other task farm services on the network. In this way, the power of multiple, heterogeneous compute servers can be combined with no effort on the part of the end user.

At this stage, ServerMeta is extremely experimental and has raised a number of interesting questions. However, it serves well as a proof of concept.

An example of the server is outlined in figure 1, showing the use of amber, beryl, Lomond and BOBCAT to produce a rendering of the Mandelbrot set using the fractalBrowse client. No modification whatsoever was made to the fractalBrowse client in order to use this service.

The meta server needs no configuration: it automatically detects the services on the network and uses all of them. This approach leads to a potential problem when there is more than one meta-server on the network. At the moment, the code will use any service that is not a ServerMeta service. However, another implementation of a meta-server, if deployed, would be used, and that might in turn attempt to use ServerMeta services. This could lead to parts of jobs getting stuck in submission loops, being passed back and forth between two or more metaservers and never being evaluated. It may on occasion be desirable to make a meta-meta-server, which combines other meta-servers together - for example, all of the devices at one site could be placed together under a meta server, with an institution-wide meta-server which brings together all resources over all sites.

I believe that manual configuration could more easily prevent such loops occurring.

An alternative approach would be for each meta-server to maintain a list of the service IDs of all of its component servers and of all of their components and so on. A meta-server would then not use a server which would ultimately pass the work back to itself.

Additional questions are raised about the handling of the various service attributes. Attributes such as number of processors can be easily produced by summing the attribute over each component service, but it is not entirely clear how to handle `SupportedAPIEntry`. Two obvious approaches are to take the union of the each set of APIs and to take the intersection of each set.

Taking the union would lead to difficulties because there is at present no way of specifying which APIs are used by the client. The metaserver would not know which particular services should be used for any given run.

Taking the intersection would solve the above problem. If a client requests a metaservice, it only needs APIs provided by every component. However, this is likely to limit the range of APIs advertised by the metaservice.

8.1 Structure

ServerMeta runs as both a `TaskFarmService` server and as a client of all of the other task farm services that it can find. It uses `ServiceFinder` to maintain a list of `TaskFarmServices`. When a job is to be executed, it takes the list of known services, removes all instances of ServerMeta, and divides the tasks up evenly between each of the remaining services. A new thread is started to handle each target service.

When a thread has completed, the results are posted to the main results vector. At this stage, `ProgressEvent` notification to the client can occur, so a client will receive all of the results from one component task farm in one burst. However, the code could be modified without too much difficulty to pass results along continuously.

8.2 Configuration and Use

Again, the usage is very similar to serverThread (section 6.2).

The server can be started with:

```
start.serverMeta
```

9 Sample Clients

Three sample clients are presented. Two perform a computation with no user interaction, whilst the third is very interactive.

9.1 Computation of π

This demonstration computes a value for the mathematical constant π by evaluating an integral which is known to be equal to π.

The evaluation is performed by dividing the integral into a number of slices (specified by `nTasks`) and treating each slice as a task. When the task is evaluated, it is cut again in finer sections (as specified by `nSlices`) and the integral evaluated at one point in each section. These results are then summed to produce the value of the integral and hence of π.

The code lives in the `jinigrid.test.piClient` package and consists of four classes: `hptPiClient`, `hpiPi`, `piAllTasks` and `piResult`. The source for these is included in Appendix A. Note that there is no new task class defined – `java.lang.Integer` is used to specify which of the evenly spaced sections is to be evaluated.

The program will use the first `TaskFarmService` that it finds.

The main class can be executed with a command such as:

```
gridclient jinigrid.test.piClient.hptPiClient 40 username
userpkfn 5 100000
```

which will result in an approximation of $\pi \approx 3.14159315$ and $error \approx 5 \times 10^{-7}$

The code deliberately contains no exception handling in order to simplify the presentation of source code. Better handling can be found in the next client.

9.2 Finding Optimal Golomb Rulers

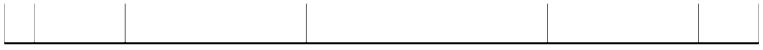


Figure 2: A seven mark Golomb ruler, with marks at spacings of 1,3,6,8,5,2

A Golomb ruler is a stick with marks on it, such that no two marks are the same distance apart as any other two marks.[6]

An Optimal Golomb Ruler (OGR) for a given number of marks is the smallest Golomb Ruler with that many marks. Golomb Rulers with many marks are known, but it is only known which are optimal for rulers of up to 23 marks. There exists a massively parallel task farm project, distributed.net, which at the time of writing is examining the 24-mark and 25-mark spaces.

The search for OGRs is implemented as a search tree, each level on the tree representing Golomb rulers of a certain length.[6] The tree is split into a number of branches, each branch forming a task. Each task produces a Golomb Ruler that is optimal in its branch and smaller than a threshold passed through the global data. The master then gathers these and locates the most optimal.

A `Ruler` class is defined, which is used both to transmit stubs to the workers and to transmit the result back to the master.

9.3 fractalBrowse

fractalBrowse is an interactive viewer for the Mandelbrot set. The user can select a region of the image to zoom into. Computation of the actual data is performed on a task farm, which

[6] In more mathematical terms, a ruler is a set $R \subset \mathbb{N}$ such that
$\forall a, b, c, d \in R \quad |a - b| = |c - d| \rightarrow \{a, b\} = \{c, d\}$

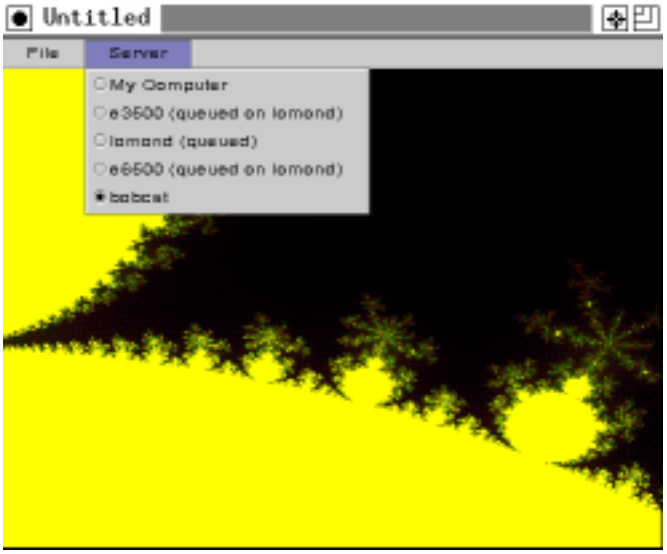


Figure 3: Screenshot of fractalBrowse, showing an activated TaskFarmSelectorJMenu

can be selected from a service menu. By default, the task farm is a single thread instance of ServerThread running locally.

As tasks come and go, the service menu updates automatically to include new services and to remove old services. If a service fails, the client will automatically and silently switch back to a local instance of ServerThread.

10 Utility classes

In this section, a number of utility classes are presented. These are used (as noted) in several of the servers and clients above.

10.1 DiscoveryWalker

DiscoveryWalker allows lookup services to be located by specifying a path, similar to the unix shell path, which names discovery groups and jini:// URLs. The path must take the form described for the `$JINIGRID_LOOKUP` environment variable, described in section 4.3.

It is an implementation of the `net.jini.discovery.DiscoveryManagement` interface, (via the `LookupDiscoveryManager` class) which provides the ability to implement custom lookup service finding code.

`DiscoveryWalker` also has some experimental code, disabled by default, which can use Lookup Discovery services. These allow a form of bridging between different lookup services.

A DiscoveryWalker is instantiated with the lookup path supplied to the constructor, and can then be used wherever a `DiscoveryManager` is needed, for example in the `ServiceFinder` or `TaskfarmSelectorJMenu` classes.

```
public class DiscoveryWalker
```

```
            extends LookupDiscoveryManager
            implements DiscoveryListener
{
   public DiscoveryWalker(String lookupPath)
            throws java.io.IOException
   \\ additionally, the DiscoveryManagement methods
}
```

10.2 ServiceFinder

ServiceFinder is based on the class of the same name presented in [5]. Its purpose is provide a simple API for clients to locate services matching a supplied template.

There are a number of constructors, which allow the client to specify varying amounts of information. The simplest takes a single parameter specifying the class/interface to be found. No attributes are required, and the contents of the `jinigrid.lookup` property are used as lookup path. More complicated constructors allow attributes and lookup path to be specified.

The `getObject()` and `getObjects()` methods return, respectively, one or all of the service objects found. If no objects have been found so far, the methods will block until at least one is found.

```
public class ServiceFinder
            implements DiscoveryListener
{
   public ServiceFinder(Class serviceInterface)
            throws IOException
   public ServiceFinder(Class serviceInterface,
            String name)
            throws IOException
   public ServiceFinder(String path,
            Class serviceInterface)
            throws IOException
   public ServiceFinder(String path,
            Class serviceInterface,
            Entry attribute)
            throws IOException
   public ServiceFinder(Class serviceInterface,
            Entry[] attributes)
            throws IOException
   public ServiceFinder(String path,
            Class serviceInterface,
            Entry[] attributes)
            throws IOException
   public Object getObject()
   public Vector getObjects()
}
```

10.3 TaskFarmSelectorJMenu

TaskFarmSelectorJMenu is a subclass of the Swing `JMenu` class. It provides a menu which lists the available task farm services as well as a "My Computer" option. As services come and go, the menu automatically updates itself.

The class provides a `getTaskFarm()` method which returns a `TaskFarm` for the currently selected service. If "My Computer" is selected, an instance of ThreadedTaskFarm configured to run on the local machine is returned.

Using this class, almost the whole of the JiniGrid service location, selection and connection process is hidden from the client program. An application uses this class by creating an instance, passing various parameters to the constructor, and adding the instance to a menu bar. When use of a task farm is required, the `getTaskFarm()` method should be called to return a `TaskFarm` object from the currently selected service.

```
public class TaskfarmSelectorJMenu
          extends JMenu
{
   public TaskFarm getTaskFarm();
   public TaskfarmSelectorJMenu(String s, int nump,
                                JarData jar);
}
```

10.4 ReliableMap

ReliableMap implements the Map interface from the Java Collections framework. It provides for the association of key and value objects, and for the retrieval of value objects based on key. The map is persistent – that is, the contents of the Map are preserved between invocations of a program.

This implementation uses ReliableLog, a utility class provided by Sun with Jini. ReliableLog provides reliable on-disk storage with atomic actions – an update to the data will either completely happen or not happen at all. This helps maintain the integrity of the data.

The body of the data is stored as a serialized HashTable, whilst updates are stored as instances of a custom inner class.

Most of the Map calls are passed through to the internal HashTable, with only the four mutating calls logged.

```
public class ReliableMap implements Map
{
public ReliableMap(String path) throws IOException

// Methods that are logged

public void clear()
public Object put(Object key, Object value)
```

```
public void putAll(Map m)
public Object remove(Object key)

// Methods that are passed straight through to the embedded Map

public boolean containsKey(Object o)
public boolean containsValue(Object o)
public Set entrySet()
public boolean equals(Object o)
public Object get(Object k)
public int hashCode()
public boolean isEmpty()
public Set keySet()
public int size()
public Collection values()
}
```

10.5 MultiVersion

This class represents a multi-part version number (for example 1.2.2) of arbitrary depth. It is used to store API versions in the SupportedAPIEntry class (section 3.6).

MultiVersion permits version numbers of arbitrary length, for example, 1.54.3.2.6. The Java Versioning Specification uses only a three part major.minor.micro version number.[2]

```
public class MultiVersion implements java.io.Serializable
{
   public MultiVersion(int[] v)
   public MultiVersion(String s)
   public String toString()
   public final int[] version;
}
```

11 Command line utilities

11.1 gridclient

Usage:
`gridclient <java options> <classname> <arguments>`

Gridclient can be used in place of the `java` command. It launches a Java virtual machine setting various JiniGrid system properties to values read from the environment.

Example:

```
gridclient jinigrid.test.ogr.MainFarm 3 50
2 user  /HitachiPT/pkfn 8
```

11.2 servicels

Usage:
`servicels -lookup jini://<hostname>/ [-verbose]`
where:
`<hostname>` is the name of a system running the Jini lookup service.
`-verbose` signifies that more information should be printed.

Servicels lists all services registered with the specified lookup service. It provides a command line equivalent for the basic functionality of the Jini Browser program.

11.3 servicekill

Usage:
`servicekill <servicename>`

Servicekill will attempt to kill the service named by `<servicename>`.

In order for this to work, the following must hold:

- The service must have a Name entry specifying `<servicename>`
- The service must be administrable, that is, it must implement the `net.jini.admin.-Administrable` interface.
- The service's administration object must implement the `com.sun.jini.admin.-DestroyAdmin` interface.

If the service is an ServerOne service, then additional code will be invoked to log onto the service before destruction is attempted.

12 User Authentication

HPT used a public/private key scheme for user authentication. The user stores his private key in a file on disk, whilst the server stores a corresponding file of user public keys. ServerOne retains that authentication mechanism.

The other servers do not have any authentication mechanism at all. Any client can connect and use them.

As it stands now, the stub code must be given read access to the private key of a user. This is bad.

In a large scale grid environment where there could be thousands or millions of users, some mechanism would need to be in place distribute user authentication information. It cannot reasonably be expected that every compute server would maintain an up-to-date copy of a centralised public key database.

The system also precludes using other authentication mechanisms based on differing levels of paranoia. For example, a site might wish to use simple passwords or might want to use a SecurID card. Neither of these approaches are accommodated with the HPT framework.

For these reasons, I chose not to implement any authentication to begin with, with the intention of maybe looking at the problem later. Unfortunately, I have not had enough time to do so.

13 Future Development

- More flexible class loading. If we are using remote events, we must have an http server set up to transmit the event listener stubs to the event producer. It would be nice if the client had the option of using this http server rather than a jar file.
  I have investigated the possibility of using a Map from `String` classname to `byte[]` bytecode. However, in this abstract setting, it is not clear how one should assign the `CodeSource` property of loaded classes, needed for code permissions.
- PipeCommsWrapper in HPT is very inefficient
- The remote events in the API could be enhanced - for example, an event could be posted when a queued job starts to run, or when a server changes status.
- Reporting of server status - the way HPT did it was in some ways backwards; the server should supply a status object; in HPT the client supplied a status object which the server updated - this precludes it being a remote object.
- How difficult would this be to convert this into a general purpose compute server, rather than just a task farm?
  It would be difficult to make a completely generic remote parallel server, but specific models could fairly easily be produced. For example, a message passing compute service could be defined, with an abstract message passing library which could use threads or MPI or whatever was available on the server.
  In the end, one might end up with an array of different service types: task farms, message passing, shared memory multithreaded, single threaded, with servers implementing as many of these models as they could. There could also be "adaptor" services (like metaservers) that would re-expose a message passing server as a task farm server.
- How do we pick which is the best server to use? A number of metrics could be developed based on expected speed, expected time until job completes, etc, perhaps based partly on the JavaGrande benchmarks.
- If we are transmitting the results with the events, it is wasteful to transmit the results at the end as well – but we may have missed some events and so missed some results. How can we check this?
  Sometimes we run out of memory on the task farm - if we are progressively transmitting results back to the client, then we can free the memory used by the results on the server – this doesn't necessarily transfer the same memory use to the client – the client can process the result and discard it.
- An optimisation to try on serverThread: give each thread its own results vector and merge them together synchronously at the end, then we don't need to synchronise on the results vector and so maybe it will be faster...

A Source of hptPiClient

A.1 hptPiClient.java - the main class

```
package jinigrid.test.piClient;

import HPT.*;
import java.io.*;
import java.util.Vector;
import jinigrid.util.*;
import jinigrid.spec.*;
import java.rmi.*;

public class hptPiClient
{

  public static void main (String args[]) throws Exception
  {
    System.out.println("JiniGrid Pi computation");

    System.setSecurityManager(new RMISecurityManager());

    TaskFarmService taskFarmService=null;
    ServiceFinder sf=new ServiceFinder(TaskFarmService.class);
    taskFarmService=(TaskFarmService)sf.getObject();

    System.out.println("Got a taskFarmService");

    JarData jar = new JarData("piClient.jar");

    TaskFarm taskFarm = taskFarmService.getTaskFarm(args[1],
      args[2],Integer.parseInt(args[3]),jar);


    int i;
    int nTasks;
    double calcPi=0;
    Vector vOfTasks;
    piAllTasks piCommonData;
    Vector results;
    nTasks=Integer.parseInt(args[0]);
    vOfTasks=new Vector(nTasks);

    for(i=0; i<nTasks;i++)
      vOfTasks.addElement(new Integer(i));
```

```
    piCommonData=new piAllTasks(
      1.0/(nTasks*Integer.parseInt(args[4])),
      Integer.parseInt(args[4]) );

    System.out.println("Running tasks...");

    piResult taskRes;

    results=taskFarm.runTask(vOfTasks,piCommonData,
      "jinigrid.test.piClient.hptPi");

    System.out.println("Run completed");

    for(i=0,calcPi=0;i<results.size();i++)
    {
      taskRes=(piResult)results.elementAt(i);
      calcPi+=taskRes.getResult();
    }

    calcPi=calcPi/((double)nTasks)
      /(double)Integer.parseInt(args[4]);

    System.out.println(
      "Calculated Pi= "+calcPi+", JVM PI= "+Math.PI);
    System.out.println(
      "Error (wrt Math.PI)="+Math.abs(Math.PI-calcPi));
    taskFarm.finish();
    System.out.println("The end.");
    System.exit(0);
  }
}
```

A.2 hptPi.java - the Slaveable implementation

```
package jinigrid.test.piClient;

import HPT.*;

public class hptPi implements Slaveable
{
  int task;
  piResult result;
  piAllTasks commonData;

  public void setTask(Object task)
  {
```

```
    this.task=((Integer)task).intValue();
  }

  public void setInitAllTasks(Object initAllTasks)
  {
    commonData=(piAllTasks)initAllTasks;
  }

  public Object getResult()
  {
    return result;
  }


  public void run()
  {
    double res = 0;
    double h = commonData.h;

    for(int t=task*commonData.samples;
        t<task*commonData.samples+commonData.samples;
        t++)
      res += ( f(h * ((double)t - 0.5)));

    result=new piResult(res);
  }

  double f(double a)
  {
    return(4.0/(1.0+a*a));
  }
}
```

A.3 piAllTasks.java - the global data

```
package jinigrid.test.piClient;

class piAllTasks implements java.io.Serializable
{
  final public double h;
  final public int samples;

  public piAllTasks(double h, int samples)
  {
    h=h;
```

```
    samples=samples;
  }

  public double geth()
  {
    return h;
  }

  public double getsamples()
  {
    return samples;
  }
}
```

A.4 piResult.java - the returned result

```
package jinigrid.test.piClient;

class piResult implements java.io.Serializable
{
  double taskSum;

  public piResult(double sum)
  {
    taskSum=sum;
  }

  public double getResult()
  {
    return(taskSum);
  }
}
```

References

[1] *bsub(1) man page from LSF*, August 1998.

[2] Sun Microsystems Inc. The Java Versioning Specification. http://java.sun.com/products/jdk/1.2/docs/guide/versioning/spec/VersioningSpecification.html.

[3] Sun Microsystems Inc. Dynamic code downloading using RMI, Guide to Features – Java Platform v1.2. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/codebase.html, 1999.

[4] Fernando Elson Mourão. A Middleware Environment for Parallel Java on the Hitachi SR2201. SSP report, Edinburgh Parallel Computing Centre, September 1998.

[5] Scott Oaks and Henry Wong. *Jini(tm) in a Nutshell*. Nutshell series. O'Reilly & Associates, 2000.

[6] William T. Rankin. Optimal golomb rulers: An exhaustive parallel search implementation. Master's thesis, Duke University, Department of Electrical Engineering, December 1993.

Benjamin Clifford
I have completed three years out of four towards the degree of MSci Mathematics at Queen Mary and Westfield College, University of London.

Supervisors:
Scott Telford, EPCC
Martin Westhead, EPCC