Decentralized Grid Services on P2P Networks: A Case Study

F R E D R I K S Ö D E R S T R Ö M

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-05

Decentralized Grid Services on P2P Networks: A Case Study

FREDRIK SÖDERSTRÖM

Examiner
Assoc. Prof. Vladimir Vlassov
(IMIT/KTH)

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-05

Abstract

Computers have been connected to networks for a long time. Traditional networks usually provide only simple services. To keep up with the ever-increasing demand for computing resources, like processing power and storage, there is a need to leverage more power from existing networks. One way of managing all resources of large networks, and letting multiple organizations share these resources with each other, is called Grid computing.

In this thesis, we examine one of the services that is necessary for a Grid, namely resource discovery, a mechanism for finding the available resources on a network. Resource discovery services are often designed to rely on some kind of central repository where all resources must be registered. But this approach does not work well in very large networks, because the central repository will become a bottleneck. Resource discovery can also be decentralized, and we suggest that it should be built on peer-to-peer technology to achieve maximum scalability. Using JXTA, a peer-to-peer platform, and the Globus Toolkit for Grid services, we create and evaluate a prototype implementation of a distributed discovery service.

Acknowledgments

I would like to thank my father, Håkan Söderström, for helping me with a number of bugs and reading drafts of the report. I would also like to thank all the programmers that have created the tools that I use every day, and have depended on for this project, especially Debian GNU/Linux and OpenOffice.org.

Table of Contents

List of Figures

1 Introduction

Two resource-sharing environments are currently competing for attention. One is peer-to-peer (P2P) computing [33] and the other is Grid computing [20]. Until recently, these technologies have been considered very different, because although they basically address the same problem – sharing a large set of resources in a coordinated manner – they do so from very different angles. In traditional networks, there is a clear separation between servers and clients, while in both P2P computing and Grid computing, all computers can act as clients and servers in the network, possibly simultaneously.

1.1 P2P Computing

P2P has traditionally been seen as a “grass-roots” technology, being developed in a non-standard way. There are many P2P networks, but each one usually has its own protocol, so they do not work well together. The first well-known P2P network was Napster, which provided (highly controversial) file sharing. Napster was shut down for legal reasons around March 2001, but this did not stop the development, and today there are countless numbers of P2P networks. Most of them are still focusing on file sharing.

To avoid some of the legal hazards, the file sharing networks often avoid using central servers, and try to hide the identities of their users. Some networks, like Freenet [18], actually have anonymity as their primary objective. The users and their resources come and go, so the availability of the individual nodes is generally low. These characteristics have given P2P a somewhat bad reputation. But this has changed in recent years, and P2P has turned into a major research topic.

P2P networks can broadly be classified in the following three categories (the exact definitions vary, though), which can also be seen as technology generations as development has progressed. However, it should be noted that the more recent generations have not necessarily superseded the earlier ones – each has its own uses, and they complement each other. The categories are:

1. Networks that rely on central servers for coordination;
2. Decentralized networks, based on flooding of messages;
3. Structured networks.

1.1.1 First-Generation Networks

Napster was a typical representative of the first generation of P2P networks, relying on central servers for coordination. When the central servers were shut down, the network could not be used, even though the files were still there – there was no way of finding them. But for special applications requiring only a moderate amount of resources for coordination, this design is viable, for example Seti@home [14] and the increasingly popular BitTorrent [19].

1.1.2 Second-Generation Networks

The second-generation networks have removed the need for central servers. Instead, they rely on **flooding** for finding things in the network. Every node has a set of neighbor nodes. When a node wants to find a resource in the network, it sends a message to its neighbors, and these in turn forward the message to their neighbors, so the network is flooded by requests. This process goes on until either the resource is found, or the request becomes outdated. All requests have a specified lifetime, or time-to-live (TTL) that is decreased every time a request is forwarded in the network.

Completely unstructured networks do not scale well, because as the network grows, more and more bandwidth is used for maintenance (for example keeping track of neighbors) rather than useful purposes. The most important second-generation network is Gnutella. It has also served as a basis for various design improvements, like in [16], which introduces a dynamically adapted overlay topology.

1.1.3 Third-Generation Networks

One way of improving the performance of second-generation networks is using **supernodes**, as in Kazaa [13]. Nodes with the highest uptime and largest available bandwidth become supernodes, and are connected to

other supernodes, forming a network of their own. Other nodes connect to a supernode instead of directly to each other. So the supernodes serve the same purpose as central servers, but in a distributed and more reliable way.

Another way of improving P2P networks consists of organizing the nodes into a structured overlay network and using distributed hash tables (DHTs), discussed below.

1.2 Overlay Networks and Distributed Hash Tables

An overlay network is a virtual network built on top of another network. The underlying network need not necessarily be a physical network; it could just as well be another overlay network. The Internet could be considered the most well-known overlay network. An overlay network provides routing, guaranteed object location, load balancing, a self-organizing structure, and, perhaps most importantly, very fast (usually logarithmic-time) lookup. Some well-known overlay networks include CAN [29], Chord [35], Pastry [30], and Tapestry [38].

A related concept is that of distributed hash tables (DHTs). A DHT requires an identifier space (usually a range of integer numbers), which is divided into smaller parts, one for each participating node. This partitioning can be used as buckets for a standard hash table which becomes distributed, hence the name. The DHT can then provide a lookup method that efficiently determines which node is responsible for a certain key.

The problem with DHTs is how to divide the identifier space. Whenever a node joins or leaves, buckets must be updated to make sure that the entire identifier space is covered, and that there are no overlapping parts. Usually the DHT only modifies buckets close to the area where the network has changed, which decreases the cost of the operation, but also creates suboptimal partitionings. And when the buckets change, data stored at the nodes may have to be moved as well. So DHTs are maintenance intensive, especially in very dynamic networks.

1.3 Grid Computing

The speed of CPUs has certainly increased over the last decade, but the speed of networks and the storage capacity has increased even faster. This trend is likely to continue, so there is a need to use the existing processing power more efficiently. For example, CERN's Large Hadron Collider [12] is expected to produce petabytes (10^{15} bytes) of data in 2006 [20]; to be able to store and, more importantly, process this data, a new approach is required.

With the increase in network speed, communication will become more or less free, so we should start to work in new, communication-intensive ways. We should be able to use all the processing power available within an organization. But quite possibly, the resources of a single organization will not be enough, requiring organizations to cooperate. When organizations start sharing resources using policies that specify what is shared and under what conditions, they become a virtual organization (VO) [23][24]. The purpose, scope, size, and duration of VOs may vary considerably, but they still share a set of requirements, for example control over shared resources and flexible sharing relationships.

So, it would be convenient if processing power (as well as other computer resources) were as available and easy to use as electrical power grids. This is where the term “Grid computing” originates. Contrary to P2P, Grid computing has been developed by research institutions and major corporations like IBM, and so has automatically gained a certain level of credibility. However, it has also become quite a buzzword lately. Some existing tools can be seen as simpler first-generation Grids, while others are simply called “Grid” for marketing reasons.

There are many different definitions of what a Grid is, and what it is not. In [21], Ian Foster, who is considered the father of Grid computing, lists three criteria that can be seen as requirements for Grids. According to Foster, a Grid is a system that:

1. Coordinates resources that are not subject to centralized control;
2. Uses standard, open, general-purpose protocols and interfaces;
3. Delivers nontrivial qualities of service.

Grids can also provide file sharing, but, compared to P2P networks, typically consist of a much smaller number of more advanced resources like supercomputers or scientific instruments. These resources are usually connected by high-speed networks, have high availability and dedicated maintenance staff. The resources are shared in a structured way, with policies for who can use what and when.

Instead of the anonymity of P2P users, Grid users are authenticated before they can use the Grid, and they may not be authorized to use all resources. However, there is still a difference when it comes to trust – current Grid users are often researchers that can be assumed to behave well, while P2P networks are designed for users that cannot be trusted.

1.4 Merging P2P and Grids

In summary, we have two technologies approaching the resource-sharing problem from different angles. P2P provides simple services that are very scalable, and Grids provide more advanced but less scalable services. Obviously, it would be desirable to have a system that used the best parts from each technology. Despite the differences, the Grid community has realized that there are things to learn from P2P, mostly concerning scalability and dynamism. The P2P community can, in turn, benefit from the research that has been done to improve Grid technology [36].

Eventually, there may not be much difference between a Grid and a P2P network. Ian Foster claims that P2P and Grid computing will converge into a common technology [22]. This will certainly require more research and development, but is indeed an interesting prospect.

1.5 Project Specification

Next, we will discuss the motivation for this project, along with a definition of the problem we are approaching. We will also outline the goals we will try to achieve.

1.5.1 Background

Today's Grids usually consist of a small to moderate amount of resources from a small number of organizations. Grids can certainly be useful on this

level, but the true potential of Grid computing will only be achieved when Grids become much larger, perhaps approaching the size of today's largest P2P networks. Managing such a network of resources is obviously very difficult. Although the P2P networks have their own technical problems (bandwidth requirements being perhaps the largest), they have shown that it is indeed possible to connect huge amounts of computers in a sensible way.

1.5.2 Expected Results

The main purpose of this Master thesis project is two-fold: (1) Study of related work on Grid and Web services, (structured) overlay networks and their use in P2P applications; (2) Development, implementation and evaluation of at least one of the Grid services specified in Open Grid Services Architecture (OGSA) (such as storage management, searching and indexing, group services, and data distribution) based on an overlay network infrastructure. The main features of the Grid service to be achieved are good scalability and low-cost self-organization.

We expect that development and evaluation of a Grid service on an overlay network will help evaluate whether such networks are useful and convenient (easy to use) for Grid services, as well as help evaluate other properties of the network that might be useful for Grid services.

In order to evaluate a Grid service implemented on top of an overlay network, we plan to develop an evaluation strategy (evaluation parameters, experimental framework and benchmark applications) and to perform evaluation experiments to estimate scalability and performance of the service. If time allows, we intend to develop a simple analytical model of the service that can allow prediction of service characteristics for different design choices at the first stages of the design, to simplify development and implementation of the service.

Expected results of this project include but are not limited to:

1. A survey of approaches to emerging Grids, Web services and P2P computing capabilities; and related work towards implementation of OGSA.

2. An architecture (structure, interfaces, algorithms and protocols) and a prototype (a reference implementation) of a Grid service as a P2P application based on an overlay network.
3. A set of design issues that must be considered in developing a Grid service as a P2P application – derived from 2.
4. An evaluation procedure (evaluation parameters, an experimental framework and benchmark applications) and results of evaluation experiments.

1.5.3 Problem Definition

The problem can be divided into three smaller tasks:

1. Finding a suitable Grid service to implement on top of a structured P2P network;
2. Creating an architecture and a prototype implementation of the service;
3. Implementing a prototype and evaluating the architecture.

The first task is finding a Grid service to improve. It was decided that resource discovery (also called resource location) was suitable. Resource discovery deals with finding the resources (compute power, storage space, data, etc) that are available in a Grid. Ian Foster, among others, claim that this service is suitable for P2P technology [26][27][28].

In Grid environments, resource discovery has traditionally been more or less centralized, requiring all participating nodes to register their resources at some kind of server that keeps track of all available resources, like the first-generation P2P networks. This works well when the Grids are not too large and perhaps consist of resources from only one organization. In this case, the organization can make sure that the resources deliver the desired qualities of service.

But sharing resources within a single organization on a relatively small level is decidedly different from handling the needs of very large virtual organizations. If we want to connect millions of nodes, or connect resources from hundreds of organizations, the centralized resource discovery model becomes troublesome. Maintaining the required servers

will become very difficult and expensive, and it is unlikely that any central authority will gain the trust of all users. Here, the P2P way of dealing with resource discovery comes in handy.

1.5.4 Architecture and Implementation of a Prototype

The second task is deciding about the architecture of a prototype resource discovery service for Grids that uses P2P technology, and then implement it. In [26], four requirements for an efficient resource discovery mechanism are listed:

1. Independence of central, global control;
2. Scalability;
3. Support for intermittent resource participation;
4. Support for attribute-based search.

The first point, independence of global control, may not be obvious at first sight. Today's Grids are being managed centrally, with administrators allowing, or disallowing, users access to various resources. Again, this works well when the Grids are small or perhaps mid-size, but it will not be practical for the large-scale Grids that are being envisioned. There is a departure from today's Grid towards the decentralized approach of P2P. For an example of this development, see CAS, described in section 2.5.2.

The first three of these requirements are more or less inherent in good P2P implementations, but the fourth requirement is different – it is not present in current P2P solutions. Also, using a global naming scheme (used in many resource discovery solutions) with attribute-based search is difficult, if at all possible.

From the strong position that the Globus Toolkit (GT) – a set of software components for creating Grid services – holds, it is obvious that the Grid service should be as interoperable as possible with GT. It must be a proper Grid service in itself, but it would also be desirable to integrate it with various GT tools as much as possible.

1.5.5 Evaluation of the Prototype

The third and final task is the evaluation of the prototype. The benefits of using decentralized resource discovery should be particularly visible in large-size Grids that are difficult to simulate, requiring estimations to be made. A number of evaluations are performed in [26] and [28], and these could serve as a good starting point. If time allows, a simple analytical model that can predict service characteristics depending on design decisions could be created, but developing the model is out of the scope of this thesis.

1.6 Thesis Overview

The remainder of this thesis is organized as follows:

In Chapter 2, a survey of related technologies is presented.

In Chapter 3, an introduction to working with the Globus Toolkit is given.

In Chapter 4, a design for solving the problem outlined above is created.

In Chapter 5, the implementation of the design from the previous chapter is described.

In Chapter 6, the implementation is tested and evaluated.

In Chapter 7, the work that has been done for this thesis is summarized.

2 Survey of Relevant Technologies

As Grids are comprised of a large number of heterogeneous resources, there is a clear need for open standards to define how the resources should interact and behave. The major standard for Grid computing is OGSA (Open Grid Services Architecture) [24]. As it builds on top of Web services, understanding the basics of Web services is fundamental.

Before we can create any Web or Grid services for this project, we need a P2P network to serve as a base for our services. We decided to use JXTA, which will be discussed next.

2.1 JXTA

JXTA [11] is a set of open protocols for P2P networking. It was originally created by Sun Microsystems, so usually the Java (J2SE) implementation of JXTA is used. However, there are implementations for other languages, for example C and Python [10], and a version for J2ME suitable for PDAs and cell phones. The JXTA protocols are also designed to be independent of transport protocols. The term JXTA is short for “juxtapose,” because P2P can be considered a juxtaposition of the traditional client/server systems.

The JXTA protocols standardize the manner in which peers:

- Discover each other;
- Organize into peer groups;
- Advertise and discover services;
- Communicate with each other;
- Monitor each other.

The JXTA software architecture consists of three layers: (1) Platform layer, (2) Services layer, and (3) Applications layer.

The platform layer includes the basic building blocks for P2P networks, like discovery, transport, creation of peers and peer groups, and security.

The services layer contains additional services, for example searching, indexing, and file sharing.

The applications layer contains the actual applications that use JXTA. These applications can be services themselves for other applications, so there is not always a clear separation of the services layer and the applications layer.

In the following subsections, a few important JXTA concepts that are relevant to this project will be introduced: Peer, peer groups, pipes, advertisements, and the discovery service.

For a more thorough description of these concepts, as well as some of the more advanced features of JXTA, see [9]. However, it should be noted that [9] is 1,5 years old, and obviously changes have been made to JXTA since it was written. For example, it claims that peers that do not find any rendezvous peers become rendezvous peers themselves automatically; this is currently not the case.

Later, we will discuss some of the problems we encountered using JXTA. It should also be noted that what we describe here is the standard building blocks of JXTA. Naturally, these can be extended to provide additional functionality, and as the source code is available, even the basic behavior of JXTA can be modified.

2.1.1 Peers

A JXTA network, like every other P2P network, consists of a number of interconnected nodes, or peers. Usually the peers are normal computers, but it can be anything that implements one or more of the JXTA protocols, for example PDAs or advanced cell phones. Each peer is uniquely identified by a peer ID.

There are four kinds of peers:

- Minimal edge peers only send and receive messages, and do not cache advertisements or help routing messages for other peers.
- A full-featured edge peer is the “standard” kind of peer. Most peers in a network are likely to be of this kind.
- Rendezvous peers behave like the standard peers, but also forward discovery requests to help other nodes to discover resources.

- Relay peers maintain information about routes to other peers, and help routing messages in the network.

Individual peers can provide various services, called peer services. Peers usually discover each other on the network to create relationships called peer groups, described next.

2.1.2 Peer Groups

A peer group is a collection of peers that are somehow related to each other. A peer group may be open to any peers to join, but it can also be restricted to a limited set of peers, depending on the purpose of the group. Initially, all peers join the so-called Net peer group, which is a default group that is created when a JXTA network is started. Peers are free to join any number of groups they desire. A peer can be a member of more than one group at the same time. The peer groups are ordered into a hierarchical structure, where each group has a “parent” group.

The JXTA protocols describe how to publish, discover, join and monitor groups, but they do not decide about when or why peer groups are created.

Peer groups provide services called peer group services. There is a basic set of services that all peer groups must provide (or use the default implementations provided by the Net peer group), but each group can also create its own services for specific purposes. The core peer group services include:

- Discovery – for finding resources such as peers and pipes (see below for more details);
- Membership – for accepting or rejecting new members of the group;
- Access – for validating requests, in terms of credentials;
- Pipe – for creating pipes (communication channels) between peers;
- Resolver – for sending generic query requests to other peers;
- Monitoring – for monitoring other members of the group.

2.1.3 Pipes

Pipes are communication channels used by peers to send messages to other peers. Pipes are usually asynchronous and unidirectional. The endpoints of the pipe are called input pipe (the receiving end) and output pipe (the sending end), respectively. There are two modes of communication: **point-to-point**, where one input pipe and one output pipe are connected, and **propagate**, where one output pipe is connected to many input pipes.

2.1.4 Advertisements

In JXTA, all resources (like peers, peer groups, and pipes) can be described in terms of XML documents called advertisements. These advertisements are published using the discovery service in JXTA. Peers discover resources by searching for advertisements. There are a number of different advertisement types, describing peers, peer groups, and pipes. Two other important advertisement types are the module class advertisement, which provides basic information about a service in the network, and the module specification advertisement, which provides more details about a service.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:MCA>
<jxta:MCA xmlns:jxta="http://jxta.org">
        <MCID>
                urn:jxta:uuid-EE123F318F184D6BBF83BA41E4AB64FE05
        </MCID>
        <Name>
                JXTAMOD:FSGRID_STORAGE
        </Name>
        <Desc>
                storage
        </Desc>
</jxta:MCA>
```

Figure 1: A JXTA advertisement

Figure 1 shows a typical JXTA advertisement. It is a module class advertisement (hence the jxta:MCA) for a storage service. First are some initial XML tags, and then comes an ID, the name of the service, and a description of the service.

2.1.5 The Discovery Service

The discovery service in JXTA is used heavily in the software created during this project, so it warrants a closer look. It uses JXTA's Peer Discovery Protocol (PDP) to discover any published resources. The J2SE implementation uses a combination of multicast to the local subnet and rendezvous peers for network crawling.

Both P2P “clients” and “servers” use the discovery service. Clients use it to get the available advertisements, either remotely, with a request sent over the network, or locally, from a cache. Servers use the discovery service to publish their services. Services can either be published directly, sending the advertisement over the network immediately, or indirectly, waiting for rendezvous peers to find and forward the advertisement.

2.2 Web Services

Web services is a distributed computing paradigm that is built on a foundation of simple, Internet-based standards like XML and HTTP. Web services are independent of programming languages and system software. This flexibility makes Web services ideal for various kinds of application integration. Web services provide methods to discover available resources and to obtain descriptions of such resources. Web services have been widely adopted. There are a number of tools and standards that help development of Grids, relieving developers from tedious tasks. Here is one of the obvious advantages of basing Grid computing on existing technology, rather than creating everything from the ground up.

There are a number of standards that have been specified by the W3C and other organizations, a few of which will be briefly described below, namely SOAP, WSDL, WSIL, and UDDI.

2.2.1 SOAP

Simple Object Access Protocol (SOAP) [8] defines an XML-based way to exchange structured data. It provides a messaging framework that is extensible and independent of underlying networking protocols and the programming model being used. To ensure good interoperability, standard protocol bindings are necessary; the SOAP 1.1 specification contains a binding for

HTTP, which is the protocol that is most frequently used with SOAP. Its text-based nature (XML) makes SOAP-based applications easier to debug. HTTP is also easier to use with firewalls than traditional binary protocols.

The root element of a SOAP message is called **envelope**. It contains an optional **header** element and a **body** element. The body contains the message payload, and the header contains information for processing the payload.

For encoding of data types, SOAP uses XML Schema. This makes it easy to specify new data types.

2.2.2 WSDL

Web Services Description Language (WSDL) [17] is used for describing and locating a Web service using an XML document. It specifies the location of the service and the methods it provides, so typically the WSDL description of a service contains everything that is necessary to use it. There are four major elements of a WSDL document:

- `portType` – the methods of the Web service
- `message` – the messages used by the Web service
- `types` – the data types used by the Web service
- `binding` – the communication protocol used by the Web service

We will encounter these elements again in GWSDL, the Grid-adapted version of WSDL, which is discussed in section 3.2.1.

2.2.3 WSIL

Web Services Inspection Language (WSIL or WS-Inspection) [15] provides conventions for locating service descriptions published by a service provider. A WSIL document can contain service descriptions that are usually URLs pointing to WSDL documents, but it could also be a reference to an entry in a UDDI registry (see below for a short introduction to UDDI).

2.2.4 UDDI

A Universal Description, Discovery, and Integration (UDDI) [7] registry service is a Web service that contains information about various services. UDDI is used by service providers to advertise their services. Service users

can UDDI to find suitable services, and get the additional metadata they need to use the services. Contrary to WSIL, UDDI uses a centralized model with repositories to keep track of the data. WSIL and UDDI can often be used together to get the best of both worlds.

2.3 OGSA

Now it is time to have a look at the major standard for Grid computing, Open Grid Services Architecture (OGSA). We will give a short overview of its architecture and highlight a few important concepts.

2.3.1 Architecture of OGSA

OGSA has a layered architecture with the following four layers:

1. Resources;
2. Web Services + OGSI Extensions;
3. OGSA Architected Services;
4. Grid Applications.

The resources can be either physical (like servers) or logical (like databases). Grid services are extended Web services, and the OGSI extensions to standard Web services are detailed below. The OGSA Architected Services layer provides services that are useful for all Grid applications, such as:

- Service management (installation, maintenance, etc);
- Service communication;
- Policy and security management;
- Job scheduling;
- Data services.

Domain-specific services can be added to this layer as well.

2.3.2 Services in OGSA

In OGSA, everything is represented by a service – a network-enabled entity that provides some capability through the exchange of messages. Compute

power, programs, databases etc, are all virtualized into services, or, more specifically, Grid services. A Grid service is a Web service extended with service data, notifications etc, as described below. Every Grid service implements one or more interfaces, which are called `portTypes` in WSDL.

As everything is modeled as services, there will be some persistent services, but also transient service instances. Database queries, data transfers and reservations of processing power are typical transient service instances. This means that service instances can be extremely lightweight entities.

2.3.3 Service Data

A Grid service, or, to be more specific, a Grid service instance, can have a set of structured data associated with it, called service data. The service data is a set of XML elements called service data elements (SDEs). Service data can be queried and retrieved through the `findServiceData` method. Generally, the service data is either **state information** (for example intermediate results of a computation) or **service metadata** (for example the cost of using the service).

2.3.4 Service Identifiers

To keep track of the service instances, every instance is assigned a unique identifier, the Grid Service Handle (GSH) when it is created. However, Grid services may be upgraded during their lifetime, so the protocol- or instance-specific information for each GSH is collected into a Grid Service Reference (GSR). A GSR contains all the information required to interact with a service instance. It is usually a WSDL document. More than one GSR can be associated with each service instance. It should be noted that having a valid GSR does not guarantee access to a service instance – for example, the service instance may have failed since the GSR was created.

2.3.5 Life Cycle Management

We must make sure that when an instance is no longer needed, its resources are reclaimed. The lifetime of a Grid service instance is handled via **soft state management**. Every service instance is assigned a specified

lifetime when it is created. When this time runs out, the instance is terminated, unless another service has sent a **keepalive** message, indicating that it wants to continue using this instance for some more time. Soft state protocols are both resilient to failure (a lost message does not cause much harm) and simple to use, because no reliable discard is required.

2.4 OGSI Extensions to Web Services

The Open Grid Services Infrastructure (OGSI) [37] extensions deal mainly with the fact that Grid services have state information, called service data, and that their lifetimes vary a lot, from transient service instances to very long-lived ones. There is a project called WSRF (Web Services Resource Framework) [6] that aims to substitute OGSI and, eventually, make Grid services converge with Web services. The next major version of the Globus Toolkit (see section 2.5) will include support for WSRF, but for now, the OGSI extensions are used for Grid services.

OGSI specifies a number of WSDL `portTypes`, or interfaces, for Grid services. Here are a few important `portTypes`:

- `GridService` – A basic `portType` that all services must implement. This is analogous to the Object class in Java, encapsulating the root behavior of the component model.
- `Factory` – A factory is a pattern where a Grid service instance is used by a client to create another, new Grid service instance. The factory returns a GSH.
- `Notification` – The notification `portTypes` are used to deliver messages between services. There is a `NotificationSource portType` for services that wish to send messages, a `NotificationSink portType` for receiving messages, and a `NotificationSubscription portType` to handle the relationship between a source and a sink. As the only exception to the rule, a service that implements the `NotificationSink portType` does not have to implement the `GridService portType`.
- `ServiceGroup` – An interface for representing a group of services.

2.5 The Globus Toolkit

During a supercomputing conference in 1995, 11 high-speed research networks in the U.S. were temporarily connected. A set of protocols was developed to allow users on this new network to run applications on computers across the country. This experiment was successful and the research continued, which led to the Globus Toolkit (GT) [5] version 1.0 being released in 1998. At the time of this writing, the current GT version is 3.2.1, but version 4.0 is scheduled to be released in January 2005.

GT is an implementation of OGSI. It is an open-source collaboration backed by the Globus Alliance [4]. There are other implementations of OGSI, but GT is the de facto standard, being used in (and developed by) both academia and corporations.

GT is not only an implementation of OGSI. It has grown to become a large collection of software that is useful for constructing Grids. The toolkit is divided into six major components:

1. Core – basic infrastructure for building Grid services;
2. Security – various security tools;
3. Data Management – tools for file transfers and replica location;
4. Resource Management – remote job submission and control;
5. Information Services – resource discovery and collection of service data;
6. XIO – a single API for all Grid I/O protocols.

These components can be used either independently or together to develop applications.

2.5.1 Core

As its name implies, the Core component is a set of building blocks that is essential to all Grid applications, offering support for soft state management, inspection, notification, discovery etc. There is also some security infrastructure and certain system-level services, for logging, management, and administration. For developers, there are some code generation tools that can be used to speed up the development of new services. For more in-

formation about the core, see [31]. Figure 2 shows the architecture of the GT Core.

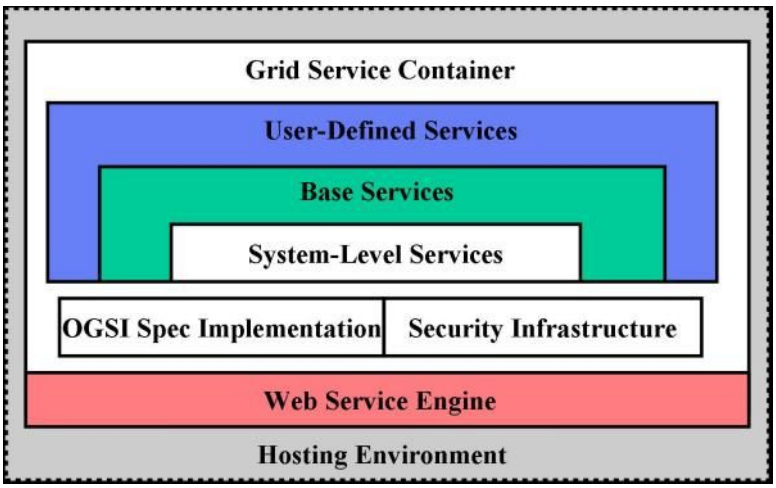

Figure 2: The GT Core architecture.

Core components have a white background.

From http://www-unix.globus.org/toolkit/docs/3.2/core/key/index.html

2.5.2 Security

GT uses the Grid Security Infrastructure (GSI) for providing secure authentication and communication over an open network. GSI helps to support security across organizational boundaries, and provides single sign-on for Grid users. It uses, and in some cases extends, well-known security standards.

GT version 3.2 introduces a new part of the security component, Community Authorization Service (CAS). CAS lets resource owners create coarse-grained policies for how their resources are used, letting the community handle the fine-grained access control and day-to-day management tasks. This is an interesting step in the development towards more decentralized Grids.

2.5.3 Data Management

The Data Management component consists of three subcomponents:

- GridFTP – A transfer protocol that is based on FTP. A number of extensions to FTP have been created to meet the requirements of Grid services.
- RFT – The Reliable File Transfer Service (RFT) is a service for controlling and monitoring GridFTP file transfers.
- RLS – The Replica Location Service (RLS) is used for data replication. It is currently in “alpha” status, suitable only for testing.

2.5.4 Resource Management

The Globus Resource Allocation and Management (GRAM) provides an interface for requesting and using various resources in a Grid. Clients can submit, monitor and shut down jobs remotely. GRAM is situated above local control and access mechanisms, and below applications and higher-order services.

2.5.5 Information Services

The Monitoring and Discovery Service (MDS) is the most important part of the information services component. It provides a generic framework for aggregation of service data and a soft state registry of available resources.

2.5.6 XIO

The goal of Extensible IO (XIO) is to create a single API for all Grid IO protocols. In distributed programming, many different protocols and APIs may be used for IO operations. With XIO, developers get access to a simpler API that is also efficient and easy to extend with new protocols.

3 Working with the Globus Toolkit

This section provides an overview of working with the Globus Toolkit. We will focus on the requirements for creating the services developed within this project. For a more detailed description, see the Globus Programmer's Tutorial [34]. The tutorial also provides a script that makes it easy to deploy Grid services. This script requires that special package names be used.

3.1 Installation and Configuration

Installing and setting up GT for simple development is not so hard. The official installation guide [3] is available at the Globus web site [4]. Our experience of the installation process, along with some hints, is collected on a web page [25].

3.2 Creating a Simple Grid Service

Creating a simple Grid service is a five-step process. The first three steps deal with describing and implementing the service, while the last two deal with its deployment:

1. Define the GWSDL interface;
2. Implement the service;
3. Configure the WSDD deployment descriptor;
4. Create a GAR file;
5. Deploy the service.

3.2.1 Define the GWSDL Interface

GWSDL is a Grid-adapted extension of WSDL. It is used to describe Grid services, in terms of the methods they provide. Note that the next version of WSDL is likely to include the GWSDL extensions, which will make GWSDL superfluous. Creating descriptions of complex `portType`s requires

some knowledge of WSDL and XML Schema, but working with primitive types is fairly straightforward.

WSDL/GWSDL is language-neutral, but at some point the interface must be referenced from a particular language. This is done using stub classes, a kind of helper classes, to make it easier for the developer. Working directly with WSDL and SOAP all the time would be very tedious. The stubs are generated automatically from WSDL descriptions by a GT tool, but we need to tell this tool where to deposit the stubs. This is accomplished using a file (called **namespace2package.mappings**) that maps GWSDL namespaces to Java packages.

3.2.2 Implement the Service

Implementing a basic Grid service in Java is not at all difficult. The script we use to deploy the service requires that the service implementation file be placed in a particular Java package. The implementation class must extend the `org.globus.ogsa.impl.ogsi.GridServiceImpl` class, which provides basic functionality for the Grid service. It must also import a `portType` interface that is generated dynamically by the build script. Finally, it must import `java.rmi.RemoteException`.

As noted above, the implementation class extends `GridServiceImpl`. It must also implement the `portType` interface for the service. All public methods must throw `RemoteException`. These are the only requirements for the simplest Grid service implementation class.

3.2.3 Configure the WSDD Deployment Descriptor

Suppose, the two most important parts of a Grid service (the service interface and the implementation) has been created. These pieces must be put together and made available through a Grid services-enabled web server. This is called deployment. The deployment descriptor describes the Grid service to the Grid service container that will host it. It is written in the WSDD (Web Service Deployment Descriptor) format.

3.2.4 Create a GAR File

All the files that have been created so far, and a number of other files, must be collected into a Grid Archive, or GAR file. Creating a GAR file involves several steps:

- Converting the GWSDL into WSDL;
- Creating stub classes from the WSDL description;
- Compiling the stubs;
- Compiling the implementation class;
- Organizing all the files into a specific directory structure.

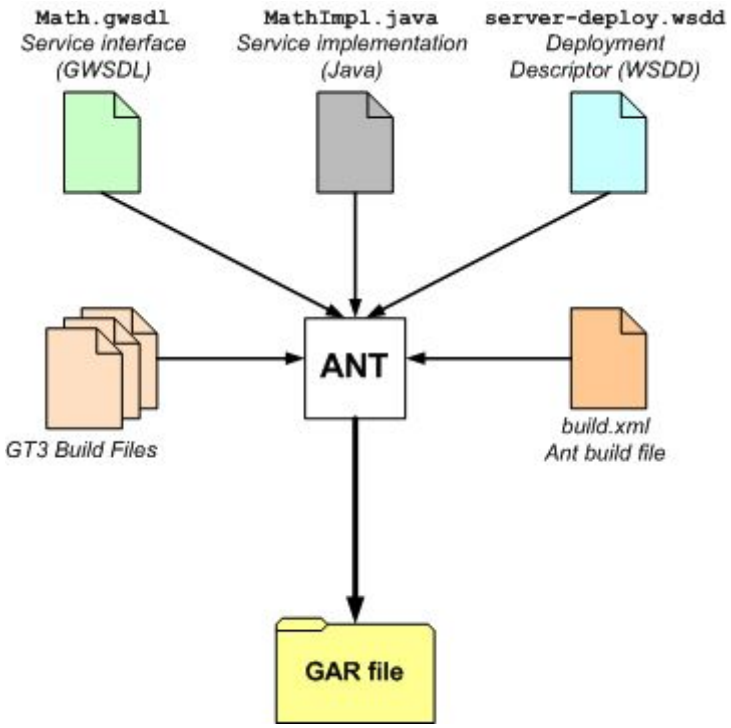

Figure 3: Creating a GAR file with Ant.

From http://www.casa-sotomayor.net/gt3-tutorial/multiplehtml/ch03s04.html

Doing this by hand would be almost infeasible, but the process can be automated using Apache Ant [2], a build tool for Java, as shown in Figure 3. All the steps listed above can be achieved by calling the Globus Programmer's Tutorial script, which in turn instructs Ant what to do. Running the script will create a GAR file for the service.

3.2.5 Deploy the Service

Deploying a GAR file is once again accomplished using Ant, and is very simple. If everything has gone well so far, the service is now ready to be used when the service container has been started.

3.3 Creating a Grid Service Client

Now that we have created a Grid service, we need some kind of client to access this service. Again, the client implementation class needs to be placed in a specific package. The client needs to import two stub classes that have been generated by Ant; the `portType` class and a service locator class. The locator class will return an instance of the `portType` class when provided with the address of the Grid service. This instance can then be used like any normal object.

3.4 Operation Providers

The service we described above extends the `GridServiceImpl` class. Being forced to extend a particular class can sometimes be problematic, especially when an existing class that already is a subclass is to become a Grid service. Fortunately, there is a simple solution to this problem: implementation by delegation, or, as it is called in GT, operation providers.

When using operation providers in GT, the deployment descriptor is used to tell the service container that the basic service functionality is still provided by `GridServiceImpl`, but we will not be forced to extend this class. Instead, we implement the `org.globus.ogsa.OperationProvider` interface. The implementation by delegation approach also leads to a more modular design, because operation providers can be plugged into many different services. A good example of this is the math service that is developed in the Globus Programmer's Tutorial. The service implementation itself only provides addition and subtraction, but existing libraries that provide other operations, for example trigonometry or matrix algebra, could easily be plugged in to extend the functionality of the original service.

To implement an operation provider, some extra work is required. We need to set up a namespace, a list of the methods the service provides, and an

object that implements the `org.globus.ogsa.GridServiceBase` interface. As noted above, when deploying the Grid service, the deployment descriptor must be modified slightly to show that the implementation class is now an operation provider. However, the GWSDL interface remains the same, so the difference will not be noticed by clients, and hence the same client can be used to access the new service.

3.5 Service Data

Service data is a very important concept in Grid services, so we will have a brief look at how to work with it. As noted above, service data is a structured collection of information that is associated with a Grid service. The service data consists of service data elements (SDEs). It should be noted that all Grid services have some basic service data, even if the service developer does not specify any service data for a service.

Using GWSDL, we can associate SDEs to a `portType`. The data type and cardinality of each SDE is specified using the XML Schema language. If there are SDEs with complex data types (anything else than the standard int, String etc), JavaBeans will be created by Ant that lets us access these complex SDEs. Also, the GWSDL description of the service and the deployment descriptor need to be updated slightly when custom service data is used.

To work with the service data on the server side, we first create an SDE object. Then we need to create a `portType` object and set its initial values, and add this object to the SDE object. Finally, the SDE object is added to the service data set of the service. This is shown in Figure 13 (page 45). When interacting with the service data, we use get/set methods on the `portType` object we created earlier.

To work with service data on the client side, we can use the service locator object to find the service data. The information we get from this object must be converted using some helper classes before it can be used by the client. When these conversion operations have finished, we have a `portType` object that we can work with exactly like before.

3.6 Creating a UI with the Globus Service Browser

Above, we have outlined how to create simple text-based Grid service clients. Another way of interacting with a Grid service in a more user-friendly way is by using the Globus service browser. This is a tool that lists all the available services of a service container, and lets the user interact with them. For each service, the service developer needs to create a class that provides access to the specific methods of the service. The service browser will add interaction with properties that are common to all Grid services, for example life cycle management.

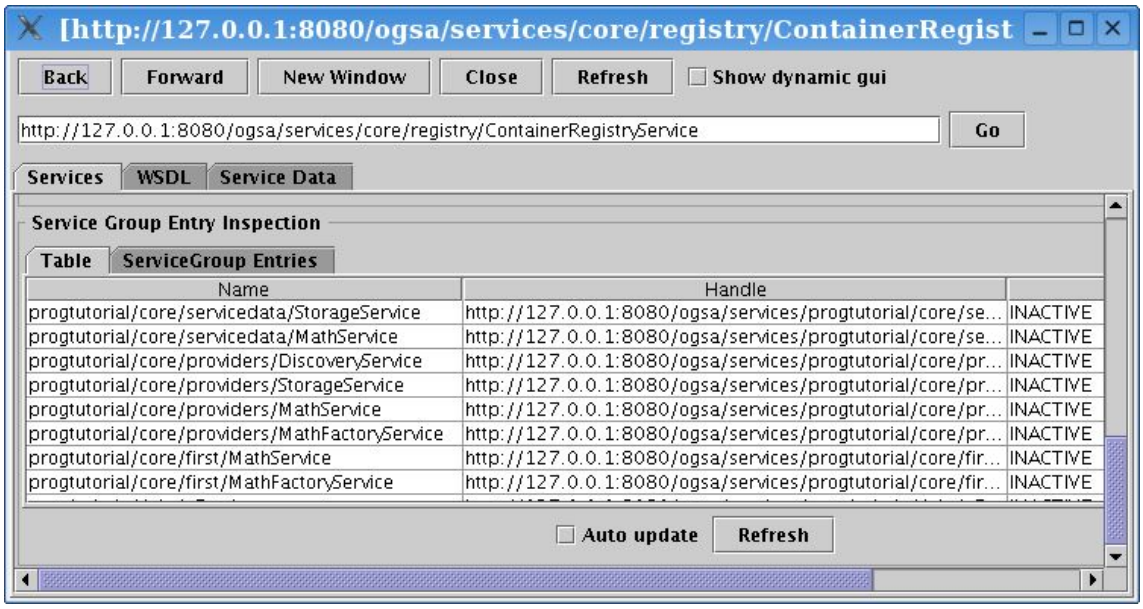

Figure 4: The Globus service browser

Figure 4 shows the service browser as it looks when it is first started. At the top there is a list of buttons for working with the windows of the service browser. The middle section (not shown in the figure) contains a number of tools for working with the services. At the bottom is the list of services running on the current container. The first one is our storage service. Below it, our discovery service, an older version of the storage service and a number of other services can be seen. Double-clicking on a service in this list will activate it and show its user interface.

Creating a class that integrates with the service browser is straightforward. The only requirement is that it extends the class

`org.globus.ogsa.gui.AbstractPortTypePanel`. The user interface is created in the same way as a standard Swing interface, and the code for interaction with the service is the same as in the text-based client described above. One thing to keep in mind, though, is that it seems like the layout cannot be controlled directly, always defaulting to a `BorderLayout`. Within the `BorderLayout`, the layout can be controlled indirectly by using panels.

Before the interface can be used, a line must be added to the **client-gui-config.xml** file in the Globus base directory, describing which `portType` is being used and which class that implements the user interface.

4 Analysis and Design

This section gives an overview of how our system should behave, and introduces the design of our solution. The design describes the P2P components, how these are adapted to work within a Grid context, and what the user interface may look like. Finally, everything is put together to create two Grid services that use P2P. We have assumed that the system will have only human users, but it should also be able to support computer users without major modifications.

4.1 Overview of the System

We are about to create a P2P-based discovery service for finding the available resources of a Grid. This means that the discovery service will connect to a P2P network (in our case realized by JXTA) to find information about the resources. Obviously, this also requires the resources to publish information about themselves on the same network. So at first sight servers (nodes that provide resources) and clients (nodes that use services provided by servers) seem to be two separate parts of the system. But one of the hallmarks of P2P is that all nodes can act as both clients and servers, and this applies to this project too, so the client side and the server side of the problem actually have very much in common.

When users want to find resources, they will start the discovery service and provide it with some search criteria, like a name or description of the service they want to use. The discovery service will connect to a JXTA network. This network could be either a global, open network or a local, private network (like a company intranet). Advertisements are published in predetermined peer groups to delimit their scope and the network load, so the discovery service will join a suitable peer group and try to find some relevant services. When it has found some interesting services, it will present a list of these services. Alternatively, if no services are found, the user gets an error message. From this list, the users should be able to find all the information they need for using their desired service.

When programmers create services that the discovery service should be able to find, they must somehow publish their presence. This could be

achieved by calling an advertisement service with a description of the service. The advertisement service will then connect to the same JXTA network as the discovery service, join a peer group and publish an advertisement of the service. The advertisement has a certain TTL, so it must be republished occasionally.

4.1.1 How the System Can Be Used

As a concrete example of a typical Grid service that could benefit from our discovery service, we will design a simple storage service. This storage service will be a completely standard Grid service. It does have one unusual feature, though: it uses code from our discovery service to advertise itself (this capability should become a Grid service of its own, as discussed in section 4.6). These advertisements can be found by our discovery service. The design of the storage service is described in section 4.6.1.

Because of the distributed nature of our discovery service, it is likely to scale better and have higher availability. It does not rely on any kind of central repository that will become a bottleneck as the system grows. Instead, the “repository” is spread over a P2P network. Hopefully, these properties will indirectly improve other services that use it, like our storage service.

After this conceptual overview of the system, we will now describe the design of the system in some more detail.

4.2 The Two Main Parts of the System

The system can be divided into two main parts, the P2P part and the Grid part. The P2P part, which uses JXTA as underlying overlay network, is responsible for:

- Setting up the P2P network;
- Creating peers;
- Organizing peers;
- Publishing advertisements for services;
- Finding published advertisements.

The Grid part builds on top of the P2P part to adapt it to the requirements of Grid services. The Grid part adds:

- Handling of service addresses (GSHs);
- Connecting to services.

As the purpose of the project was to build a kind of P2P base for Grid services, the Grid part obviously depends on the P2P part. However, it is desirable for the Grid part to be as loosely coupled to the P2P part as possible, to make it more flexible. Using the P2P part should introduce only minor, if any, modifications to existing Grid services. The choice of JXTA for basic P2P functionality has forced certain design choices, but these should only be a concern for the P2P part; the networking technology should be transparent to the Grid part. During the initial development, it was also found that it was very useful to let the P2P part be “stand-alone” so that it could be tested on its own.

Given solid background knowledge and experience, the best way of designing this system would probably be to list the requirements of the Grid part, and then design the P2P part accordingly. However, due to lack of said knowledge and experience, we started with the P2P part.

4.3 The P2P Part

JXTA was chosen as a provider of basic P2P technology, partly because it works well with Java, and partly because it seemed mature (it has existed for about 3,5 years, a long time in the P2P world). There are certainly other interesting P2P overlay networks, some of which probably have better performance than JXTA, but for this prototype, ease of development was more important than maximum performance. For a more complete implementation, another overlay network could be used, without changing the Grid part too much, although most of the P2P part would have to be rewritten in this case.

A P2P network consists of peers that act as both clients and servers, so it was obvious that there would be some functionality specific for clients and some for servers, while some functionality is common to both kinds of peers. So, the basic design consists of an abstract class called *`Node`* that contains the code necessary for both clients and servers, and two classes

called *Client* and *Server* that extend the *Node* class with specific functionality for clients and servers, respectively (Figure 5).

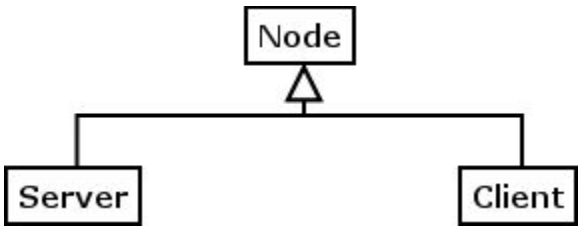

Figure 5: The P2P classes

Because of uncertainty about how a user interface may be integrated with the GT tools, and for flexibility in general, it was decided that these classes should not implement a user interface of their own. Instead, the user interface is provided by subclasses. This proved to be a good idea, which will be discussed later.

4.3.1 The Node Class

As noted above, the `Node` class contains the functionality that is common to both clients and servers. It should be able to:

- Start up and initialize the JXTA network;
- Handle discovery of various kinds of advertisements;
- Create a peer group if none is found and set up its services;
- Join a peer group.

There are ways of creating secure peer groups in JXTA, where the user has to provide user name and password etc to join the group. For simplicity, security issues have been ignored in this project.

4.3.2 The Client Class

The `Client` class should be able to:

- Send requests for relevant advertisements in a peer group, using the JXTA discovery service;
- Handle incoming advertisements about relevant services;
- Present the available advertisements to the user, or

- Handle the case where no advertisements are found;
- Get input from the user, to find out which service to use;
- Contact the server the user has selected.

Note that the presentation of advertisements and handling of input from the user is to be delegated to UI subclasses that implement methods specified in this class.

4.3.3 The Server Class

The `Server` class should be able to:

- Create and publish advertisements for its service, using the JXTA discovery service;
- Communicate with clients that want to use the service.

Classes that provide a user interface for a `Server` only need to print messages in some way.

4.3.4 Communication between Clients and Servers

As described above, clients and servers are obviously supposed to be able to communicate with each other. The standard way of doing so in JXTA is by using pipes (see section 2.1.3). The JXTA `Server` class provides a pipe for communication, but it is only used for testing the P2P part. It is not used in Grid environments.

In JXTA, all messages must have a so-called **tag** and an optional **namespace** identifier. The tag that must be used is specified in a class called `FSGridConstants` that provides some common constants for this project. The messages in this system do not use namespaces. Apart from the tag, no special protocol is used.

4.4 The Grid Part

The Grid part of the system is responsible for making the P2P part usable for Grid services. The main classes for the Grid part are `GridClient` and `GridServer`. As shown in Figure 6, they extend the JXTA `Client` and `Server` classes (shown in Figure 5), respectively.

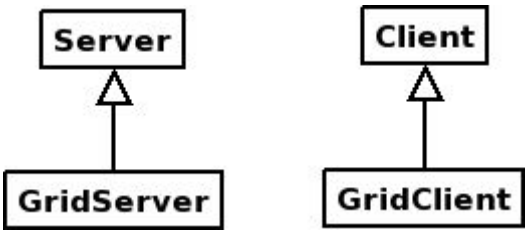

Figure 6: The main Grid classes

4.4.1 The GridServer Class

The `GridServer` class is fairly similar to the `Server` class. A GridServer can store a GSH, or address, of the service it provides. It also handles the JXTA user name and password. This is usually provided by the user when a JXTA-based application runs, but the `GridServer` class must be usable without user input or command line arguments when it is running within the service container.

4.4.2 The GridClient Class

Unlike the `GridServer` class, the `GridClient` class is decidedly different from the regular JXTA `Client` class. The `GridClient` needs to:

- Send a message to the server;
- Get a reply from the server in the form of an address of the service;
- Connect to this address and do something useful;
- Handle the case where no advertisements are found.

Most of this is implemented in other classes, so the `GridClient` class is in fact very small. The first two requirements are mostly handled by the `Client` class, and the last two are to be fulfilled by Grid service clients that use this class for access to the JXTA part.

The `GridClient` class handles the JXTA user name and password in the same way as the `GridServer` class, described above.

4.4.3 The ServiceConnector Interface

Grid service clients that wish to use the `GridClient` class need to implement an interface called `ServiceConnector`, that has two methods, one for

connecting to a service, and one for handling the case where no advertisements are found.

4.5 The User Interface

The user interface is provided by the `TextClient`/`GridTextClient` and `TextServer`/`GridTextServer` classes, which in turn get their functionality from the classes `ServerTextUI` and `ClientTextUI` (Figure 7).

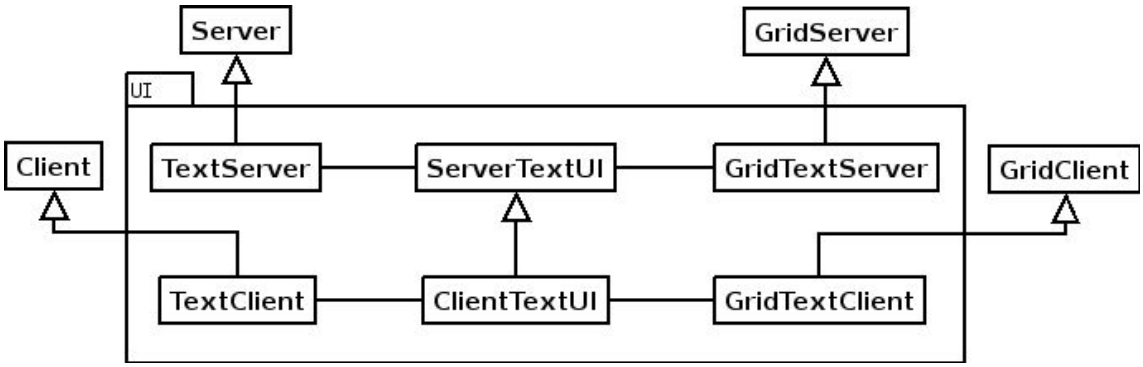

Figure 7: UI classes

Separating the functionality into so many different classes may make the design seem a little cluttered at first, but in fact it helps to increase flexibility and avoid code duplication.

Early in the project, a simple GUI was developed for testing of the JXTA part. However, this GUI was not updated to work with the Grid part, because we found that a better way of providing a GUI would be to use the Globus service browser, which was described in section 3.6. This is a tool that lists all the available services that are running in some container. The user can click on any of the services to work with them. The service browser provides access to the properties that are common to all Grid services (life cycle management, for example), and the designer of the Grid service can add an interface for the particular needs of each service. This proved to be both easy to use and develop for, as well as much more powerful than any standalone user interface we could have created within a reasonable time. A typical user interface for a service is shown in Figure 8.

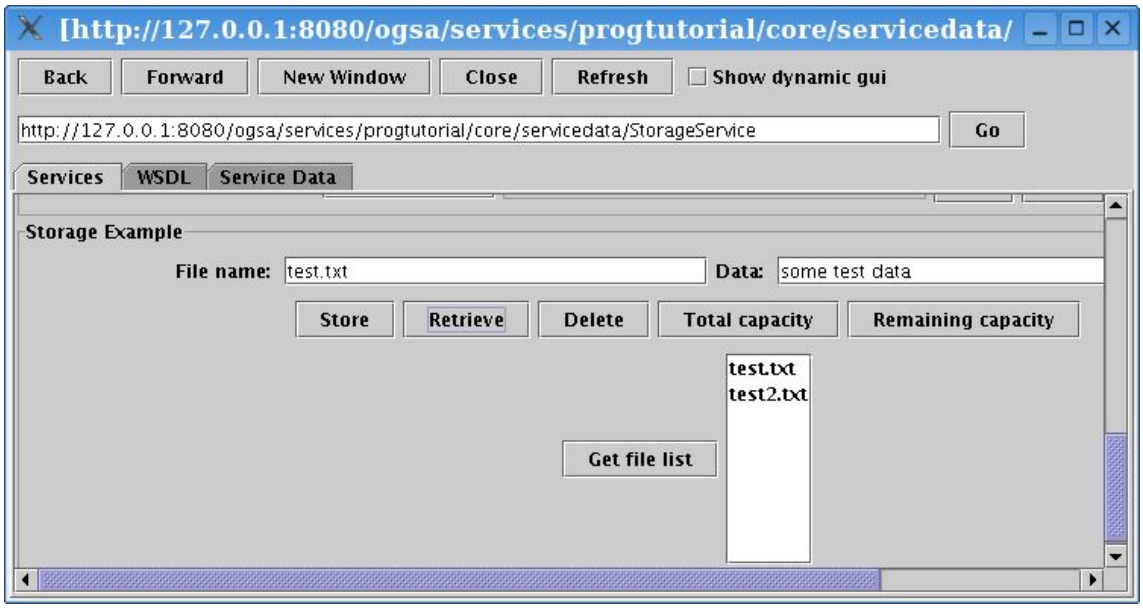

Figure 8: The storage service as seen in the service browser

In this case, it is the storage service described below. At the top there are a number of buttons for the service browser window. The middle section (not shown here) contains a set of buttons for working with general service properties. The “Storage Example” panel contains a set of widgets for interacting with the storage service.

4.6 Design of Grid Services

The design that has been described in the previous subsections might be interesting in itself, but to see if it works in practice, we need some prototype service to test the system. And, as everything in OGSA is modeled as services, making the discovery process a Grid service on its own was a natural development. There should also be a separate `AdvertisementService` for publishing advertisements, instead of services using the Grid code directly, which is currently the case. We have not created such a service yet, but it would be straightforward to do so.

Developing and deploying simple, prototype-level Grid services for GT is not very difficult, but still tedious. So it was decided that the prototype service using the existing code should be developed first, both for learning and for checking that everything worked as expected. A Grid service that is

both typical and easy to implement was needed, and it was decided that a storage service would be a good choice.

4.6.1 The Storage Service

We introduced our storage service in section 4.1.1. Now we will have a closer look at it. Let us emphasize again the fact that the service is a completely standard Grid service. The only special feature it provides is that it uses code from our distributed discovery service to advertise itself. These advertisements can be found by our discovery service. In a very large Grid, a standard, centralized discovery service is likely to become a bottleneck, which would prevent other services from being used efficiently. Using our distributed discovery service instead should indirectly improve our storage service.

We designed two versions of the storage service. The first is a very basic service that can only store and retrieve a file with hard-coded name and content. This version provided an easy way of getting used to developing Grid services. The second version is slightly more realistic, getting rid of the hard-coded values, and also providing a delete operation. In addition, three service data elements were added: total capacity, remaining capacity, and a list of available files. The files that are managed by the storage service are stored in a directory on the server the service is running on. Again, security issues have been ignored to simplify the design and development. A simple client for testing this service was also required.

4.6.2 The Discovery Service

The discovery service that we have created is very simple; it can only search for service names. It would be straightforward to add searching based on service descriptions, because descriptions of services can easily be added to JXTA advertisements.

5 Implementation

This section describes the implementation of the system.

5.1 The P2P Part

The P2P part contains of three main classes:

- `Node`
- `Client`
- `Server`

The `Node` class provides functionality that is common to both clients and servers. The `Client` and `Server` classes extend the `Node` class to add specific client-side/server-side functionality.

5.1.1 The Node Class

The first thing the `Node` class needs to do is to start up the JXTA network and initialize some basic JXTA services (discovery and rendezvous), as shown in Figure 9. This is straightforward, with the exception of the rendezvous service, which will be discussed later.

```
//create the JXTA net peer group, the group all peers
//normally join when started
try {
    netPeerGroup = PeerGroupFactory.newNetPeerGroup();
    ...
}

//get and initialize various JXTA services
discoveryService = netPeerGroup.getDiscoveryService();
discoveryService.addDiscoveryListener(this);

rdvService = netPeerGroup.getRendezVousService();
rdvService.addListener(this);
```

Figure 9: Initializing the JXTA net peer group and basic services

Then the `Node` class sets up the Grid peer group. First, it checks whether there are any advertisements for this group in the local advertisement cache. If not, it uses the discovery service to try to find one. If any advertisements are found, these are stored in the local cache. When an advertisement is found, the Grid peer group is created from the advertisement, and the node joins this group.

The `Node` class defines methods for handling the three kinds of advertisements that exist in JXTA: peer advertisements, peer group advertisements, and general advertisements (all other kinds of advertisements). Only the peer group method, as noted above, is implemented, though; the rest is left for subclasses that may need this functionality.

Finally, the `Node` class defines a method for printing simple output that subclasses providing user interfaces should override.

5.1.2 The Server Class

When a `Server` starts, its main need is to create (or recreate from a file) an advertisement for the service it provides, and then publish this advertisement. In fact, three separate advertisements are required for each service in this system. First is the module class advertisement, which only contains basic information (name and description). Then there is a pipe advertisement that is used to provide information about the pipe, or communication channel, that this server will be listening to. Finally, there is the module specification advertisement, which includes more detailed information about the service, including the pipe advertisement.

```
      //get the module class advertisement for the service
      ModuleClassAdvertisement moduleClassAdv = getModuleClassAdv(
            name, description);
       gridDiscoveryService.publish(moduleClassAdv);
       gridDiscoveryService.remotePublish(moduleClassAdv);

       //get the module specification advertisement for the service
       ModuleSpecAdvertisement moduleSpecAdv = getModuleSpecAdv(
            name, description, version, creator,
            moduleClassAdv.getModuleClassID(), specURI);

       //get the pipe advertisement for the service
      //and store it with the module spec adv
       PipeAdvertisement pipeAdv = getPipeAdv(name);
       moduleSpecAdv.setPipeAdvertisement(pipeAdv);

       //then we can publish the module spec adv as well
       gridDiscoveryService.publish(moduleSpecAdv);
       gridDiscoveryService.remotePublish(moduleSpecAdv);
```

Figure 10: Publishing advertisements

The module class and module specification advertisements are then published in the Grid peer group using the JXTA discovery service (the pipe advertisement is contained in the module specification advertisement). This is

shown in Figure 10. Finally, it creates a `JXTAServerPipe` from the pipe advertisement, and starts listening to this pipe. Clients can send messages on this pipe, and the server will reply. This communication is only for testing, though, because the server does not provide any real JXTA services. The `Server` class is just a building block for the Grid services that we will create (the Grid clients will not use the pipe).

5.1.3 The Client Class

The `Client` class uses the `Node` class to set up the basic JXTA services. Then it starts looking for advertisements that match its requirements, as shown in Figure 11. JXTA advertisements contain basic information about services, such as name, description, version, and creator. These information fields can be used when trying to find advertisements. The current implementation of the `Client` class only uses the name of the service, but it could easily be extended to look for descriptions or other criteria.

```
       //first we check the local cache for interesting advertisements
      discoveryEnum = gridDiscoveryService.getLocalAdvertisements(
             DiscoveryService.ADV, "Name", "JXTASPEC:" + serviceName);

      if (discoveryEnum != null && discoveryEnum.hasMoreElements()) {
             //an advertisement was found in the cache, so we can use
             //it immediately
             handleAdvDiscovery(discoveryEnum);
      } else {
             //the cache did not contain anything useful,
             //so we have to search remotely
             gridDiscoveryService.getRemoteAdvertisements(null,
                    DiscoveryService.ADV, "Name", "JXTASPEC:" +
                    serviceName, advThreshold, null);
      }
```

Figure 11: Client trying to find advertisements

When a suitable number of attempts to find advertisements have been made, it continues by presenting the advertisements that were found. Alternatively, if no advertisements were found, a callback method is invoked, which makes it easy for subclasses to provide their own behavior for this case. Then it waits for the user to select the desired service, and using the advertisement for this service, the appropriate server is contacted. As always, presenting advertisements and handling user input is taken care of by subclasses, not the `Client` class itself.

5.1.4 Additional Classes

The three classes mentioned above (`Node`, `Client`, and `Server`) provide the core functionality of the JXTA part, but there are a number of other classes in this part. These will be described next.

As noted above, the core JXTA classes do not provide any direct user interface. This functionality is provides by the classes `TextServer` and `TextClient`. But the way of interacting with the user is exactly the same in both the JXTA part and the Grid part, so the actual interaction code is provided by two other classes, `ServerTextUI` and `ClientTextUI`.

The interface `FSGridConstants` provides two important constants, GROUP_NAME and TAG. They define the name of the Grid peer group and the tag that marks all messages, respectively. It also defines names of a storage service and a processing power service, which have been used for testing during development.

Finally, the `ServiceDescription` class provides a way of describing a service in a friendlier way than using JXTA advertisements.

5.1.5 Problems with JXTA

We selected JXTA because we perceived it as being somewhat of an informal standard for working with P2P in Java. However, we have had major problems with JXTA during development, which we will describe next.

- As noted in our introduction to JXTA, there is a lack of good documentation. This is certainly not an unusual problem, particularly in open source projects, but it is nevertheless a big obstacle. The programmer's guide [11] contains a good description of the concepts of JXTA, but it is outdated, sometimes incorrect and the example code it provides is very limited.
- There is a lack of best practices for working with basic JXTA concepts like rendezvous peers and peer groups. The programmer's guide only describes very simple uses of JXTA functionality, which is inadequate for real applications. Certainly there are real applications built on JXTA whose source code is available to study, but they can be hard to understand, and there is no way of knowing if they

are correct. Some information can be found in mailing list archives and wikis, but it is unstructured and incomplete.

- During development, the rendezvous service has hardly ever worked. We do not know whether this is due to configuration mistakes or coding errors somewhere. After investigating this issue rather thoroughly, we decided to move on without a working rendezvous service, because it has not really hindered the development, only made it more difficult to test the system.

However, the single largest problem in this project, making JXTA run inside of Globus, is of such importance that it will be discussed separately (see section 5.4).

We have no experience working with other P2P overlay networks in Java, but we are nevertheless convinced that there must be tools available that are better than JXTA.

5.2 The Grid Part

The Grid part of the system is very compact. It consists of only five classes, none of which contains more than roughly 100 lines of code. This is possible because of the modular design of the system. The classes are:

- `GridServer`
- `GridClient`
- `ServiceConnector` (interface)
- `GridTextServer` (for text UI)
- `GridTextClient` (for text UI)

5.2.1 The GridServer Class

The `GridServer` class extends the basic JXTA `Server` class to provide some extra “Grid” functionality, namely storing the address, or GSH, of the service it provides. This value is actually stored in the `specURI` field of JXTA advertisements. It is not clear from the documentation what this field is supposed to be used for, so we have taken the liberty of using it for storing a GSH.

The `GridServer` class must also handle the JXTA user name and password that is required to run a JXTA peer. In the `Server` class, the user can specify these as command line arguments (to JXTA, not the `Server` class itself). But this will not work in the Grid service container environment, as far as we know. So, the `GridServer` instead uses system properties to read the user name and password. This behavior is deprecated in the newest release of JXTA, and it is certainly not flexible or secure, but it is a simple solution that works well within the limits of this project. Figure 12 contains a `GridServer` constructor.

```
    public GridServer(String name, String description, String GSH,
              String version, String creator) {
        super(name, description, GSH, version, creator);

        Config.setJxtaHome(".jxta2");

        try {
            System.setProperty("net.jxta.tls.principal", "jxta");
            System.setProperty("net.jxta.tls.password", "jxtajxta");

            ...
        }
    }
```

Figure 12: A GridServer constructor

In standard JXTA, the name of the JXTA configuration directory for each node is also stored in a system property. But the `GridServer` class uses our slightly modified version of JXTA (described in section 5.4) instead. Notice the line in Figure 12 that calls `Config.setJxtaHome()`, which sets the JXTA configuration directory in our version of JXTA.

5.2.2 The ServiceConnector Interface

Before we describe the `GridClient` class, we need to have a look at the `ServiceConnector` interface. This interface describes two methods that must be implemented by a class that wants to connect to a service. The first method is called `connectToService`, and, as its name implies, this method is called when an attempt to connect to a service is made. The second method is called `noAdvertisementsFound`, and it is called if no suitable advertisements are found.

5.2.3 The GridClient Class

The `GridClient` class has an extra instance variable, an object that implements the `ServiceConnector` interface. The actual code for connecting to a service, or handling the absence of advertisements, is contained in this object, and the `GridClient` just forwards everything to this object.

The `GridClient` class handles the JXTA user name, password and configuration directory in the same way as the `GridServer` class does, described above.

5.2.4 UI Classes

The classes that provide a text interface for the Grid classes (`GridTextServer` and `GridTextClient`) are very similar to the classes of the JXTA part with the same functionality. The only difference is that they extend the Grid “base” classes instead of the JXTA “base” classes. And again, the actual implementation of the user interface is delegated to `ServerTextUI` and `ClientTextUI`, respectively.

5.3 Implementation of Services

The JXTA part and the Grid part of the system, together with the Grid discovery service, constitute the foundation of this project. But to see if everything works in practice, a prototype implementation of a storage service was implemented using this system.

Creating a GT Grid service is rather straightforward in terms of development, but deploying a service is considerably harder. To simplify this process, we have, throughout the development, used a script from the Globus Programmer's Tutorial [34], as described in chapter 3. This chapter also gives an overview of working with GT.

5.3.1 The Storage Service

The storage service is implemented as an operation provider (operation providers are described in section 3.4). The main class is called `StorageProvider`. As all operation providers, it implements the `org.globus.ogsa.OperationProvider` interface. It also implements the

`org.globus.ogsa.GridServiceCallback` interface, which provides callback methods that run when a service is created, activated, etc. We use the `postCreate()` callback method to initialize the service data and start a `GridTextServer` using `JXTAClassLoader`. The service has three public methods: `store()`, `retrieve()`, and `delete()`. It also has three service data elements: `getFileNames`, `getTotalCapacity`, and `getRemainingCapacity`. The files that the storage service manages are stored in a directory on the server the service is running on.

The most complicated part of the storage service is the code that deals with service data, shown in Figure 13. Note that the code in Figure 13 is only intended to briefly show how to work with service data – the code for the storage management can be improved in many ways.

```
private ServiceData storageSDE;
private StorageDataType storage;

private GridServiceBase base;

...

//create service data element
storageSDE = base.getServiceDataSet().create("StorageData");

//create a StorageDataType instance and set intial values
storage = new StorageDataType();
storage.setFileNames(dataDirectory.list());
storage.setTotalCapacity(TOTAL_CAPACITY);
storage.setRemainingCapacity(TOTAL_CAPACITY);

//set the value of the SDE to the StorageDataType instance
storageSDE.setValue(storage);

//add SDE to service data set
base.getServiceDataSet().add(storageSDE);
```

Figure 13: Setting up service data

Currently, the code for publishing a storage service advertisement is integrated into the storage service itself, through its use of a `GridTextServer`. This should become a Grid service of its own, perhaps called `AdvertisementService`, which would let all kinds of Grid services publish their advertisements. Creating such a service would be fairly straightforward, because all the JXTA code is already created, and it just needs to be wrapped into a Grid service.

5.3.2 The Discovery Service

The discovery service implementation is very similar to the storage service. It is also an operation provider. It has two public methods: `getServices()`, and `getAddress()`. `getServices()` returns a list of all available services with a particular name, and `getAddress()` returns the address (GSH) of a service that a user has selected (this method is not working fully). The discovery service does not use any service data.

5.4 Running JXTA Inside of Globus

During this project, the discovery service and the storage/advertisement service have been developed somewhat independently. We did not run the services together until they were both working on their own without problems. This was a major mistake, because running both services simultaneously (which is obviously necessary for this project) proved to be extremely difficult.

Each JXTA node needs to use its own configuration directory, so the name of this directory must be set in some way. Usually, this information is collected from a Java system property called JXTA_HOME. When we tried to run more than one node within JXTA, the nodes failed to set their configuration directories correctly. At first, we thought the error was caused by the fact that system properties have global scope within each JVM. So we took advantage of the fact that JXTA's source code is freely available, and we modified it, removing the dependence on this system property entirely. This helped a little, but did not solve the problem entirely. Our continued investigations revealed that the problem did not really originate from the system properties, but was instead related to class loaders. Before we can discuss our solution to this redefined problem, we will introduce Java class loaders.

5.4.1 Java Class Loaders

This section gives a short overview of class loaders. For more details, see [32], a very good introduction to class loaders.

All Java classes are loaded by a class loader. Usually, this happens “behind the scenes” and programmers do not need to think about it. But there are

two cases where class loaders are important. One is when you want to load class files in some new way. The best example is a standard Java applet, which reads class files over a network instead of from a hard disk. The other case is when a larger system needs to load smaller parts that cannot be entirely trusted, for example application servers and containers.

Class loaders are organized into a hierarchy. At the top of this hierarchy is the bootstrap class loader which contains the basic runtime classes provided by the Java Virtual Machine. Then comes the system class loader which handles classes from the class path. Finally, at the lowest level are the user-defined class loaders. When a class loader is ordered to load a class, it will usually ask its parent to load it first, and if this fails, it will try loading the class on its own.

5.4.2 Class Loaders, JXTA and Globus

JXTA requires each node to be loaded by its own class loader. This is no problem when running stand-alone applications, because they run in different JVMs. Also, application servers and containers usually load their services with different class loaders to prevent them from interfering with each other, or the system itself (see the documentation for Apache Tomcat [1] for an example of this behavior). But in Globus, this does not seem to be the case.

5.4.3 The Class Loader Solution

To save us some work, we used an existing class loader from a JXTA subproject, called `PeerClassLoader`. We had to make a small but important modification to this class as well. A client returns a list of `ServiceDescription`s that describes the services it has found, and these need to be accessible by other classes that have not been loaded in the same class loader. A simple solution to this problem is to treat `ServiceDescription` as a kind of system class that will always be loaded by the system class loader.

Using `PeerClassLoader` directly is inconvenient, so we created a class called `JxtaClassLoader` that provides a simpler interface to

`PeerClassLoader`, as well as a few methods to interact with the objects (in fact, these are literally Java `Objects`) it creates.

```
        try {
            if (className.equals(CLIENT_NAME)) {
                 //set up a client
                constructorArg = new Class[1];
                constructorArg[0] = Class.forName("java.lang.String");
            } else if (className.equals(SERVER_NAME)) {
                //set up a server
                constructorArg = new Class[3];
                constructorArg[0] = Class.forName("java.lang.String");
                constructorArg[1] = Class.forName("java.lang.String");
                constructorArg[2] = Class.forName("java.lang.String");
            }

            ...

            constructor = startupClass.getConstructor(constructorArg);
           //args is an array of Objects that contain the arguments
            Object startupInstance = constructor.newInstance(args);
       }
```

Figure 14: Calling constructors with reflection

`JxtaClassLoader` provides one method for creating a new server instance (specifically a `GridTextServer`). As there currently are no ways of interacting with a server once it has started, this is the only method required for the server side of `JxtaClassLoader`. On the client side, there is one method for creating a new client (again, it is a `GridTextClient`), but also methods for running service discovery and returning information about the discovered services. These methods, and constructors for creating the objects in the first place, are called using reflection, as exemplified in Figure 14. Unfortunately, these methods do not work fully, so some more work is required to complete this part of the prototype.

Figure 15 shows a conceptual overview of the class loading system used in this prototype. It is conceptual in the sense that the system class loader is actually not only responsible for loading `ServiceDescription`, but `JxtaClassLoader` and `PeerClassLoader` as well. But this is what it looks like from a programmer's point of view.

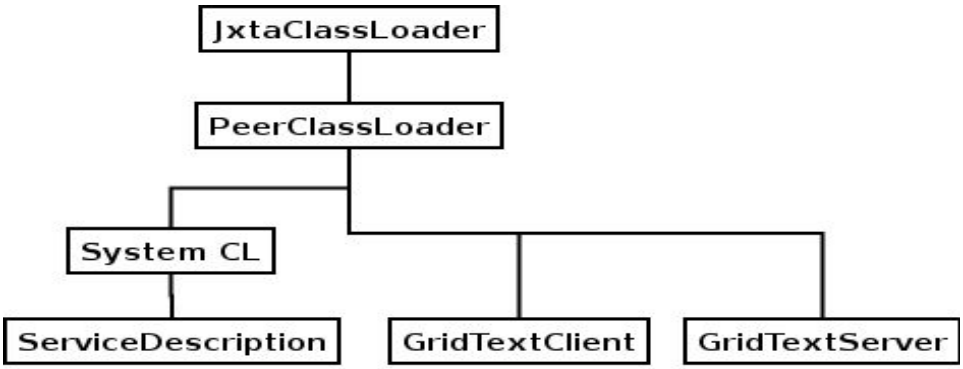

Figure 15: Class loading system

`JxtaClassLoader` is specifically tailored for our needs, so it is not a general solution.

6 Evaluation

In this section, we will run some tests to evaluate the performance and scalability of our services. We use a storage service client that uses our discovery service to find the available storage service(s). We will also study how our solution with a special class loader affects performance.

All tests described below have been ran on the same computer, with a 1.4 GHz processor, 512 MB of memory, and Debian GNU/Linux with kernel version 2.4.26-1-k7. We have used J2SE 1.4.2_02, J2EE 1.4, and GT 3.2.1. (J2EE provides some XML classes that are used by all Grid services). Each client has performed five iterations of connecting to and working with a server each time it runs.

An important fact to keep in mind when reading the numbers is that the time it takes for a client to perform its actions is highly dependent on how it is configured. This performance depends on the following factors:

- How long a client waits before giving up/continuing when searching for advertisements
- What happens when a client finds an advertisement – does it continue to search for more advertisements or does it stop searching?
- The existence of cached information. As an example, information about the peer group a client tries to join is almost always cached, but if it is not, it will take longer

Here, one must strike a balance between how much information you want to find and for how long a user will have time to wait. This certainly varies considerably. When searching for services in a huge P2P network spread around the globe, different settings must be used compared to when a local network is used. These issues affect all the numbers quoted in this section. So, these numbers should only be seen as indications of how the system performs under the circumstances we have tested.

6.1 Class Loading Overhead

As we have described in section 5.4, we were forced to use a special class loader to be able to use JXTA within Globus. We also needed to use some of

Java's reflection capabilities to work with the objects we get from the class loader. Before we can study the performance behavior of our services, we will have a look at how the class loader and reflection affects the performance. The simplest way of seeing the difference is the time it takes to start a node, either a client or a server, as seen in Figure 16. Currently, measuring the difference in runtime performance is more difficult.

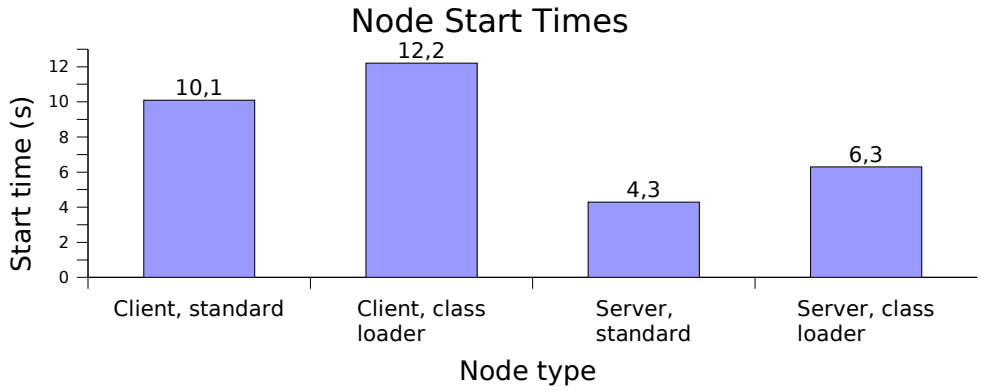

Figure 16: Node start times

These values are measured outside of Globus, running as ordinary programs. As we can see here, using our special class loader, together with reflection, adds about 20% overhead to the process of starting a client. But this value is very optimistic, because the client spends considerable time waiting for servers to reply, as discussed above. A server, on the other hand, does not need to spend as much time waiting, so the value we get here (almost 50% higher than using a standard object), is probably more realistic.

6.2 Scalability

We will now try to find out how scalable our solution is. First, let us look at the time it takes for a storage client to discover and connect to a storage service and perform a simple action, like storing a small file (Figure 17). In the tests below, a single storage service is running and used for evaluation.

We can see that when a single client is running, each iteration of connecting to a service and performing some action takes a little more than seven seconds. This time increases slowly until we run 15-20 clients simultaneously. At this point, the time starts growing more quickly. Another interest-

ing issue is at which point the server gets so busy that it is no longer able to serve all clients within the expected time (Figure 18).

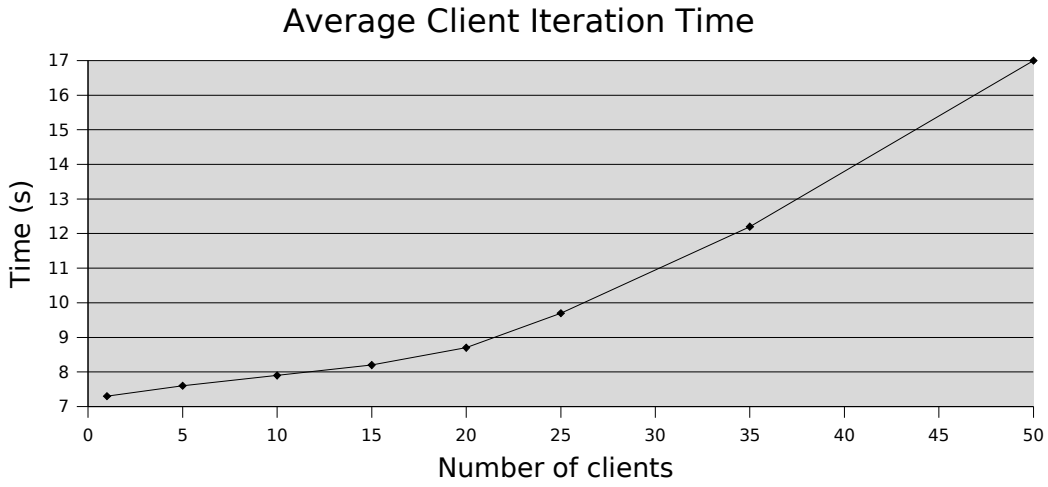

Figure 17: Average client iteration time

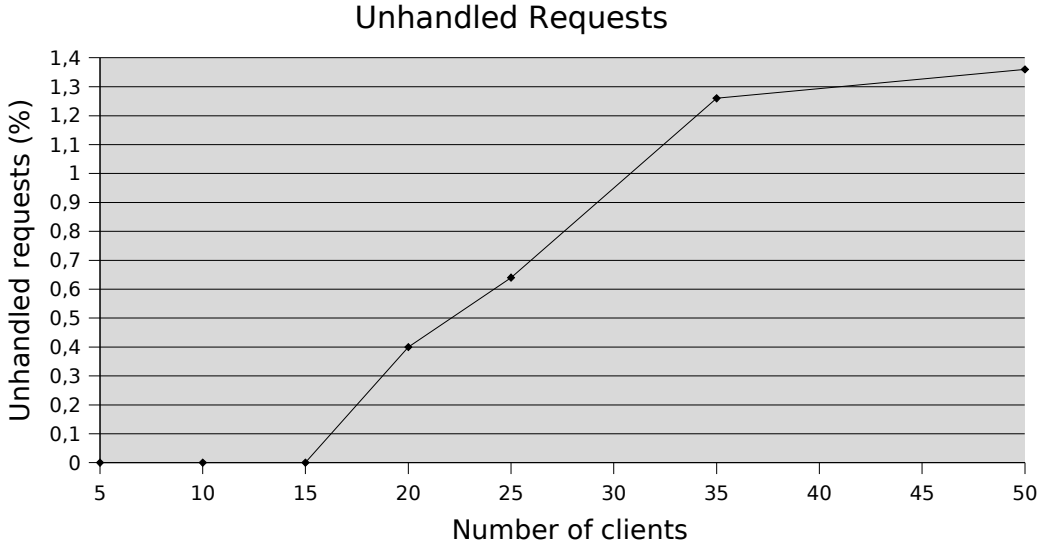

Figure 18: Unhandled requests

Similarly to the tests we ran above, the server seems to be able to handle up to 15 clients simultaneously, and serve all their requests. When we run 20 or more clients, the server is no longer capable of doing this, because the CPU load becomes too high. A certain amount of requests will not be handled in time.

7 Conclusions and Future Work

We believe that there is a great potential in Grids that approach the size of today's P2P networks, and combining these two technologies makes good sense. In this report, we have studied one aspect of such a proposed, future merger between P2P and Grid technology – a P2P-based discovery service for Grids. We have designed such a service and created a prototype implementation.

7.1 Is This a Good Idea?

When we started this project, we knew we had an interesting problem in front of us, and we were convinced that we had an equally interesting solution to this problem. Unfortunately, various problems have stopped us from examining this problem (and its solution) as thoroughly as we would have wished. This is certainly somewhat discouraging, but we see these problems as implementation issues, and they have not changed our view of the ideas behind the project. We hope that our work can inspire further investigations of this problem. We also hope that our prototype can serve as a starting point for a more complete implementation.

7.2 Technology and Problems

As noted in several sections of this report (especially section 5.4), we have faced considerable problems trying to implement and test our ideas. Some of these were not surprising – we had no prior experience working with neither JXTA nor GT. However, JXTA caused us much more trouble than we had reason to expect.

7.3 Ease of Development

Creating our first, very simple grid service and making it run correctly took us a long time. Once we learned from our initial errors, it has been surprisingly easy to create our Grid services, considering the complexity of GT. However, a few things that qualify this statement must be kept in mind. Our services are simple and cannot be compared to full-scale solutions. The class loader also makes it more complicated for clients to use our services.

The error messages produced by GT are often ambiguous and not particularly helpful, and debugging is often difficult.

7.4 Possible Improvements

There are certainly a number of ways that the software developed for this project could be improved. Here are some of the major possible improvements:

- Adding features to the discovery service. The discovery service has only the basic capability of searching for names of services. This could be extended to searching for descriptions and other properties.
- Further integration with GT. Currently, the discovery service is integrated with GT in the sense that it is a proper Grid service, and it has a user interface that works with the Globus service browser. It would be interesting to integrate the discovery service with the information tools in GT as well, to provide a more transparent discovery interface for Grid users. However, we have not attempted this, due to lack of time and documentation. Also, a separate `AdvertisementService` should be created.
- Replacing JXTA. As mentioned above, we have had quite a few problems with JXTA, so replacing JXTA with some other overlay network could provide benefits for developers. It is also quite likely that there are other overlay networks that perform better than JXTA.
- Security. Security issues have, due to time constraints, been largely ignored during this project. JXTA provides various security measures to protect peer groups, and it is also possible to modify the Java implementation to increase security. GT provides even more security-related tools.

8 List of Abbreviations

API	Application Programming Interface
CAN	Content Addressable Network
CAS	Community Authorization Service
CERN	Conseil Européen pour la Recherche Nucléaire
CPU	Central Processing Unit
DHT	Distributed Hash Table
GAR	Grid Archive
GRAM	Globus Resource Allocation and Management
GSH	Grid Service Handle
GSI	Grid Service Infrastructure
GSR	Grid Service Reference
GT	Globus Toolkit
GUI	Graphical User Interface
GWSDL	Grid Web Services Definition Language
HTTP	Hypertext Transport Protocol
J2ME	Java 2 Platform, Mobile Edition
J2SE	Java 2 Platform, Standard Edition
JXTA	“Juxtapose”
MDS	Monitoring and Discovery System
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
P2P	Peer-to-peer
PDA	Personal Digital Assistant
PDP	Peer Discovery Protocol
RFT	Reliable File Transfer Service
RLS	Replica Location Service
SDE	Service Data Element
SOAP	Simple Object Access Protocol
TTL	Time-to-live
UDDI	Universal Description, Discovery, and Integration
UI	User Interface
URL	Uniform Resource Locator
VO	Virtual Organization
W3C	World Wide Web Consortium
WSDD	Web Services Deployment Descriptor
WSDL	Web Services Definition Language
WSIL	Web Services Inspection Language
WSRF	Web Services Resource Framework
XIO	Extensible Input/Output
XML	Extensible Markup Language

9 References

[1] Tomcat Class Loading Howto. URL: http://jakarta.apache.org/tomcat/tomcat-4.1-doc/class-loader-howto.html

[2] Apache Ant. URL: http://ant.apache.org

[3] GT Installation Instructions. URL: http://www.globus.org/toolkit/docs/3.2/installation/

[4] Globus Alliance Web Site. URL: http://www.globus.org

[5] Globus Toolkit. URL: http://www.globus.org/toolkit/

[6] WSRF. URL: http://www.globus.org/wsrf/

[7] UDDI: Universal Description, Discovery and Integration. URL: http://www.uddi.org

[8] W3C: SOAP 1.1. URL: http://www.w3.org/TR/SOAP/

[9] JXTA Programmer's Guide. URL: http://www.jxta.org/docs/JxtaProgGuide_v2.pdf

[10] JXTA Core Projects. URL: http://core.jxta.org

[11] JXTA. URL: http://www.jxta.org

[12] CERN's Large Hadron Collider. URL: http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/

[13] Kazaa. URL: http://www.kazaa.com

[14] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. In Communications of the ACM, volume 45, pages 56-61, 2002.

[15] P. Brittenham. An Overview of the Web Services Inspection Language. 2001. URL: http://www.ibm.com/developerworks/webservices/library/ws-wsilover/

[16] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pages 407-418. ACM Press, 2003.

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

[18] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability, 2000.

[19] B. Cohen. BitTorrent. URL: http://bitconjurer.org/BitTorrent/

[20] I. Foster. The Grid: A New Infrastructure for 21st Century Science. Physics Today, 55 (2). 42-47. 2002.

[21] I. Foster. What is the Grid? A Three Point Checklist. GRIDToday, July 20, 2002.

[22] I. Foster, A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03), February 2003, Berkeley, CA.

[23] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications, 15 (3). 200-222. 2001.

[24] I. Foster, C. Kesselman, J. Nick, S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[25] F. Söderström. Installing, configuring and running Globus Toolkit 3.2. URL: http://www.student.nada.kth.se/~d99-fso/GTinstall.html

[26] A. Iamnitchi, I. Foster. A Peer-to-Peer Approach to Resource Location in Grid Environments. In J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinski eds. Grid Resource Management, Kluwer Publishing, 2003.

[27] A. Iamnitchi, M. Ripeanu, I. Foster. Locating Data in (Small-World?) Peer-to-Peer Scientific Collaborations. 1st International Workshop on Peer-to-Peer Systems (IPTPS'02), Cambridge, MA, March 2002.

[28] C. Mastroianni, D. Talia, O. Verta. P2P Protocols for Membership Management and Resource Discovery in Grids. ICAR-CNR Technical Report RT-ICAR-CS-04-02, March 2004.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In Proc. of ACM SIGCOMM, Aug. 2001.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In International Conference on Distributed Systems Platforms (Middleware), Nov. 2001.

[31] T. Sandholm, J. Gawor. Globus Toolkit 3 Core - A Grid Service Container Framework. July 2, 2003. URL: http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf

[32] A. Schaefer, Inside Class Loaders. URL: http://www.onjava.com/pub/a/onjava/2003/11/12/classloader.html

[33] C. Shirky. What is P2P... And What Isn't? URL: http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html

[34] B. Sotomayor. The Globus Toolkit 3 Programmer's Tutorial. URL: http://www.casa-sotomayor.net/gt3-tutorial/

[35] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the ACM SIGCOMM '01 Conference, San Diego, California, August 2001.

[36] D. Talia, P. Trunfio. Toward a Synergy Between P2P and Grids. IEEE Internet Computing, vol. 7, no. 4, pp. 94-96, 2003.

[37] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt, D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Draft Recommendation, June 27, 2003.

[38] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.

A: Data for Evaluation Diagrams

Node Startup Times

Type of node	Startup time (s)
Client, standard	10.1
Client, class loader	12.2
Server, standard	4.3
Server, class loader	6.3

Average Client Runtime

Number of clients	Average client runtime (s)
1	7.3
5	7.6
10	7.9
15	8.2
20	8.7
25	9.7
35	12.2
50	17.0

Unhandled Requests

Number of clients	Unhandled requests (%)
5	0
10	0
15	0
20	0.40
25	0.64
35	1.26
50	1.36

B: Use Cases

Use Cases for all JXTA Peers (Clients and Servers)

- The peer looks for existing Grid peer groups
- The peer looks for existing rendezvous peers

The peer looks for existing Grid peer groups

Participating actors: The peer, an arbitrary peer from the discovered peer group

Flow of events:

1. The peer sends a discovery request looking for a Grid peer group. It checks the local cache for existing advertisements.
2. An arbitrary peer from an existing Grid peer group receives the request and replies by sending the advertisement of the group.
3. The peer receives the advertisement and uses the information contained in it to join the Grid peer group.

Extensions:

1. If no old advertisements are found in the cache, the request is propagated by the discovery service.
2. If no other peer replies, the peer creates its own Grid peer group and advertises it. The use case terminates.

The peer looks for existing rendezvous peers

Participating actors: The peer, a rendezvous peer

Flow of events:

1. The peer tries to find a rendezvous peer in the Grid peer group.
2. The rendezvous peer responds, and the two peers establish a connection.

Extensions:

2. If no rendezvous responds within a certain time, the peer becomes a rendezvous peer for the Grid peer group

Use Cases for the JXTA Client

- The client looks for suitable services
- The client presents the available services to the user, and contacts the server the user selects

The client looks for suitable services

Participating actors: The client, one or more servers

Flow of events:

1. The client sends a discovery request looking for a service. It checks the local cache for existing advertisements.
2. One or more servers that provide relevant services respond with the advertisements of their services.
3. The advertisements are stored by the client.

Extensions:

1. If no relevant advertisements are found in the local cache, the request is propagated by the discovery service.
2. If no servers respond in time, the client notifies the user, and the use case terminates.

The client presents the available services to the user, and contacts the server the user selects

Participating actors: The client, a server

Flow of events:

1. The client presents the stored service advertisements to the user.
2. The user selects one of the available services.
3. Using the information in the advertisement, the client contacts the server that provides the service.
4. The server responds with an address where the actual service can be found.

Use Cases for the JXTA Server

- The server initializes its service and advertises it
- The server listens for incoming client connections

The server initializes its service and advertises it

Participating actors: The server

Flow of events:

1. The server initializes its service.
2. The servers uses the discovery service to publish the advertisement for the service.

The server listens for incoming client connections

Participating actors: The server, one or more clients

Flow of events:

1. The server listens for incoming client connections on the communication channel specified in the advertisement.
2. When the server receives a message from a client, it responds with the address of the service it provides.

C: Javadoc

Class Node

```
java.lang.Object
  └ fsgrid.Node
```

All Implemented Interfaces:
net.jxta.discovery.DiscoveryListener, java.util.EventListener, net.jxta.rendezvous.RendezvousListener

Direct Known Subclasses:
Client, Server

---

public abstract class **Node**
extends java.lang.Object
implements net.jxta.discovery.DiscoveryListener, net.jxta.rendezvous.RendezvousListener

This class represents an abstract JXTA network node or peer. It provides basic services that are common to both client and server nodes.

Field Summary	
protected int	**advThreshold** The number of advertisements that a single peer will be allowed to deliver when it has been discovered
protected java.lang.String	**description** A general description of the service that this peer provides or looks for
protected net.jxta.discovery.DiscoveryService	**discoveryService** This is the discovery service for the net peer group, the default group that all JXTA peers join when they are started.
protected net.jxta.discovery.DiscoveryService	**gridDiscoveryService** This is the discovery service for the grid peer group, the group for all grid service related peers.
protected net.jxta.peergroup.PeerGroup	**gridPeerGroup** The grid peer group is a group that all grid service JXTA peers are supposed to join.
protected java.lang.String	**name** The name of the peer, or more specifically, the name of the service that it provides or will look for

Constructor Summary

Node(java.lang.String name) Basic constructor for a grid service peer that sets its name	
Node(java.lang.String name, java.lang.String description) Basic constructor for a grid service peer that sets its name and description	

Method Summary	
void	**clearAdvCache**() This method clears the advertisement cache
void	**discoveryEvent**(net.jxta.discovery.DiscoveryEvent discoveryEvent) Handles discovery events that occur when this peer discovers other peers, peer groups and advertisements.
protected void	**handleAdvDiscovery**(net.jxta.discovery.DiscoveryEvent discoveryEvent) This method is usually called when a general advertisement (not peer or peer group advertisement) is discovered
protected void	**handleGroupDiscovery**(net.jxta.discovery.DiscoveryEvent discoveryEvent) This method is usually called when a peer group advertisement is discovered.
protected void	**handlePeerDiscovery**(net.jxta.discovery.DiscoveryEvent discoveryEvent) This method is usually called when a peer advertisement is discovered
protected void	**initializeGridPeerGroup**() Initializes the grid peer group, the group that all grid service JXTA peers are supposed to join.
protected void	**initializeRendezvous**() Initializes the rendezvous service for both the net peer group and the grid peer group.
void	**printMessage**(java.lang.String message) This method provides a way of showing output to the user in a way that is not dependent on the UI that is used
void	**rendezvousEvent**(net.jxta.rendezvous.RendezvousEvent rendezvousEvent) Handles rendezvous events that occur when this peer is contacted by other rendezvous peers.
protected void	**startJxta**() Starts the JXTA environment, and initializes basic services like discovery and rendezvous

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

discoveryService

```
protected net.jxta.discovery.DiscoveryService discoveryService
```

This is the discovery service for the net peer group, the default group that all JXTA peers join when they are started. A discovery service provides methods for discovering various things in a JXTA network, like peers and peer groups.

---

gridDiscoveryService

```
protected net.jxta.discovery.DiscoveryService gridDiscoveryService
```

This is the discovery service for the grid peer group, the group for all grid service related peers. A discovery service provides methods for discovering various things in a JXTA network, like peers and peer groups.

---

name

```
protected java.lang.String name
```

The name of the peer, or more specifically, the name of the service that it provides or will look for

---

description

```
protected java.lang.String description
```

A general description of the service that this peer provides or looks for

---

gridPeerGroup

```
protected net.jxta.peergroup.PeerGroup gridPeerGroup
```

The grid peer group is a group that all grid service JXTA peers are supposed to join. There they can privately advertise and search for grid services. Special care of synchronization is required when this variable is used.

---

advThreshold

```
protected int advThreshold
```

The number of advertisements that a single peer will be allowed to deliver when it has been discovered

Constructor Detail

Node

```
public Node(java.lang.Stringname,
            java.lang.Stringdescription)
```

Basic constructor for a grid service peer that sets its name and description

Parameters:
`name` - The name of the peer, or more specifically, the name of the service that it provides or will look for
`description` - A general description of the service that this peer provides or looks for

---

Node

```
public Node(java.lang.Stringname)
```

Basic constructor for a grid service peer that sets its name

Parameters:
`name` - The name of the peer, or more specifically, the name of the service that it provides or will look for

Method Detail

startJxta

```
protected void startJxta()
```

Starts the JXTA environment, and initializes basic services like discovery and rendezvous

---

initializeGridPeerGroup

```
protected void initializeGridPeerGroup()
```

Initializes the grid peer group, the group that all grid service JXTA peers are supposed to join. It tries to discovery any existing groups. If none is found, it creates one itself. Finally it joins this group.

---

initializeRendezvous

```
protected void initializeRendezvous()
```

Initializes the rendezvous service for both the net peer group and the grid peer group. At the moment, none of this is working very well...

---

rendezvousEvent

```
public void rendezvousEvent(net.jxta.rendezvous.RendezvousEventrendezvousEvent)
```

Handles rendezvous events that occur when this peer is contacted by other rendezvous peers.

Specified by:
`rendezvousEvent` in interface `net.jxta.rendezvous.RendezvousListener`
Parameters:
`rendezvousEvent` - the rendezvous event that has just occurred

---

discoveryEvent

`public void` **`discoveryEvent`**`(net.jxta.discovery.DiscoveryEventdiscoveryEvent)`

Handles discovery events that occur when this peer discovers other peers, peer groups and advertisements. It delegates the handling to the handleXYZDiscovery methods.

Specified by:
`discoveryEvent` in interface `net.jxta.discovery.DiscoveryListener`
Parameters:
`discoveryEvent` - the discovery event that has just occurred

---

handleAdvDiscovery

`protected void` **`handleAdvDiscovery`**`(net.jxta.discovery.DiscoveryEventdiscoveryEvent)`

This method is usually called when a general advertisement (not peer or peer group advertisement) is discovered

Parameters:
`discoveryEvent` - the discovery event that has just occurred

---

handlePeerDiscovery

`protected void` **`handlePeerDiscovery`**`(net.jxta.discovery.DiscoveryEventdiscoveryEvent)`

This method is usually called when a peer advertisement is discovered

Parameters:
`discoveryEvent` - the discovery event that has just occurred

---

handleGroupDiscovery

`protected void` **`handleGroupDiscovery`**`(net.jxta.discovery.DiscoveryEventdiscoveryEvent)`

This method is usually called when a peer group advertisement is discovered. This method is synchronized because it handles some initialization of the grid peer group.

Parameters:
`discoveryEvent` - the discovery event that has just occurred

---

printMessage

`public void` **`printMessage`**`(java.lang.Stringmessage)`

This method provides a way of showing output to the user in a way that is not dependent on the UI that is used

Parameters:
`message` - the text that should be presented to the user

---

clearAdvCache

`public void clearAdvCache()`

This method clears the advertisement cache

Class Server

```
java.lang.Object
  └ fsgrid.Node
      └ fsgrid.Server
```

All Implemented Interfaces:
net.jxta.discovery.DiscoveryListener, java.util.EventListener, net.jxta.rendezvous.RendezvousListener, java.lang.Runnable

Direct Known Subclasses:
GridServer

---

public class **Server**
extends Node
implements java.lang.Runnable

This class extends the general Node class to provide server-side capabilities, that is to provide some kind of service

---

Field Summary

Fields inherited from class fsgrid.Node
advThreshold, description, discoveryService, gridDiscoveryService, gridPeerGroup, name

Constructor Summary
Server(java.lang.String name) A basic constructor for the Server class requiring only the name of the service it provides to be set
Server(java.lang.String name, java.lang.String description) A constructor for Server where the name of and a description of the service it provides is given
Server(java.lang.String name, java.lang.String description, java.lang.String specURI) A constructor for Server where the name of and a description of the service it provides is given

Server(java.lang.String name, java.lang.String description, java.lang.String specURI, java.lang.String version, java.lang.String creator) A constructor for Server that sets all its available properties

Method Summary	
protected void	**initializeAdvertisements**() This method creates (or recreates from files) a number of necessary advertisements, and performs some other initialization
void	**run**() Is this stuff really necessary?
protected void	**startConnectionThread**() Starts the thread that listens for incoming client connections
void	**startWork**() This method initializes the JXTA environment and starts the server's work
protected void	**waitForClientMessages**(java.lang.String reply) This method waits for clients to connect to the server, and when they connect it replies with the GSH of this service

Methods inherited from class fsgrid.Node
clearAdvCache, discoveryEvent, handleAdvDiscovery, handleGroupDiscovery, handlePeerDiscovery, initializeGridPeerGroup, initializeRendezvous, printMessage, rendezvousEvent, startJxta

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Server

```
public Server(java.lang.Stringname,
              java.lang.Stringdescription,
              java.lang.StringspecURI,
              java.lang.Stringversion,
              java.lang.Stringcreator)
```

A constructor for Server that sets all its available properties

Parameters:

`name` - the name of the service this server provides
`description` - a description of the service that this server provides
`version` - the version of the service that this server provides
`specURI` - included here to cover all advertisment properties - only used by subclasses

(GridServer)
`creator` - The name of the creator of this service

Server

```
public Server(java.lang.Stringname,
              java.lang.Stringdescription,
              java.lang.StringspecURI)
```

A constructor for Server where the name of and a description of the service it provides is given

Parameters:
`name` - the name of the service this server provides
`description` - a description of the service that this server provides
`specURI` - included here to cover all advertisment properties - only used by subclasses (GridServer)

Server

```
public Server(java.lang.Stringname,
              java.lang.Stringdescription)
```

A constructor for Server where the name of and a description of the service it provides is given

Parameters:
`name` - the name of the service this server provides
`description` - a description of the service that this server provides

Server

```
public Server(java.lang.Stringname)
```

A basic constructor for the Server class requiring only the name of the service it provides to be set

Parameters:
`name` - the name of the service this Server provides

Method Detail

startWork

```
public void startWork()
```

This method initializes the JXTA environment and starts the server's work

---

startConnectionThread

```
protected void startConnectionThread()
```

Starts the thread that listens for incoming client connections

---

run

`public void run()`

Is this stuff really necessary?

Specified by:
`run` in interface `java.lang.Runnable`
See Also:
`Thread.run()`

---

initializeAdvertisements

`protected void initializeAdvertisements()`

This method creates (or recreates from files) a number of necessary advertisements, and performs some other initialization

---

waitForClientMessages

`protected void waitForClientMessages(java.lang.Stringreply)`

This method waits for clients to connect to the server, and when they connect it replies with the GSH of this service

Parameters:
`reply` - the reply the server should send to connecting clients

Class Client

`java.lang.Object`
  └ `fsgrid.Node`
    └ **`fsgrid.Client`**

All Implemented Interfaces:
net.jxta.discovery.DiscoveryListener, java.util.EventListener, net.jxta.rendezvous.RendezvousListener

Direct Known Subclasses:
GridClient

---

public class **Client**
extends Node

This class extends the general node class to provide client-side capabilities, which means finding and connecting to interesting servers

---

Javadoc

Field Summary

Fields inherited from class fsgrid.Node
advThreshold, description, discoveryService, gridDiscoveryService, gridPeerGroup, name

Constructor Summary
Client(java.lang.String name) A constructor for Client that sets the name of the service it will look for
Client(java.lang.String name, java.lang.String description) A constructor for Client that sets the name and description of the service it will look for

Method Summary	
fsgrid.ServiceDescription[]	**getCollectedAdvertisements**() Returns the service descriptions this client has collected
protected void	**getServiceAdvertisements**(java.lang.String serviceName) Sends out requests looking for interesting services, handles the incoming advertisements and prints the results
protected void	**handleAdvDiscovery**(net.jxta.discovery.DiscoveryEvent discoveryEvent) Handles incoming advertisements
protected void	**handleAdvDiscovery**(java.util.Enumeration discoveryEnum) Handles incoming advertisements
protected void	**noAdvertisementsFound**() This method is called if no appropriate advertisements are found
protected void	**presentAdvertisements**() Presents a list of advertisements that this client has collected
protected void	**printServiceList**(fsgrid.ServiceDescription[] serviceArray) Presents a list of services that this client has found.
java.lang.String	**returnSelection**(int value) This method is called when the user has made a selection from the list of services.
void	**runDiscovery**() This method runs the discovery process and presents the results
void	**runDiscovery**(java.lang.String serviceName) This method runs the discovery process and presents the results
protected java.lang.String	**sendMessageToServer**(net.jxta.protocol.ModuleSpecAdvertisement moduleSpecAdv) Sends a predetermined message to a server and returns the reply from the server
void	**startWork**() Initializes the JXTA environment and starts the client's work
void	**startWork**(java.lang.String serviceName) Initializes the JXTA environment and starts the client's work

Methods inherited from class fsgrid.Node
clearAdvCache, discoveryEvent, handleGroupDiscovery, handlePeerDiscovery, initializeGridPeerGroup, initializeRendezvous, printMessage, rendezvousEvent, startJxta

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Client

```
public Client(java.lang.Stringname,
              java.lang.Stringdescription)
```

A constructor for Client that sets the name and description of the service it will look for

Parameters:
`name` - the name of the service that this client will look for
`description` - a description of the service that this client will look for

---

Client

```
public Client(java.lang.Stringname)
```

A constructor for Client that sets the name of the service it will look for

Parameters:
`name` - the name of the service that this client will look for

Method Detail

startWork

```
public void startWork(java.lang.StringserviceName)
```

Initializes the JXTA environment and starts the client's work

Parameters:
`serviceName` - the name of the service the client will search for

---

startWork

```
public void startWork()
```

Initializes the JXTA environment and starts the client's work

---

Javadoc

runDiscovery

```
public void runDiscovery(java.lang.StringserviceName)
```

This method runs the discovery process and presents the results

Parameters:
`serviceName` - the name of the service to search for

---

runDiscovery

```
public void runDiscovery()
```

This method runs the discovery process and presents the results

---

getServiceAdvertisements

```
protected void getServiceAdvertisements(java.lang.StringserviceName)
```

Sends out requests looking for interesting services, handles the incoming advertisements and prints the results

Parameters:
`serviceName` - the name of the service the client will search for

---

presentAdvertisements

```
protected void presentAdvertisements()
```

Presents a list of advertisements that this client has collected

---

noAdvertisementsFound

```
protected void noAdvertisementsFound()
```

This method is called if no appropriate advertisements are found

---

handleAdvDiscovery

```
protected void handleAdvDiscovery(net.jxta.discovery.DiscoveryEventdiscoveryEvent)
```

Handles incoming advertisements

Overrides:
`handleAdvDiscovery` in class `Node`
Parameters:
`discoveryEvent` - a discovery event

---

handleAdvDiscovery

```
protected void handleAdvDiscovery(java.util.EnumerationdiscoveryEnum)
```

Handles incoming advertisements

Parameters:
`discoveryEnum` - an enumeration of discovered advertisements

---

sendMessageToServer

```
protected java.lang.String sendMessageToServer
(net.jxta.protocol.ModuleSpecAdvertisementmoduleSpecAdv)
```

Sends a predetermined message to a server and returns the reply from the server

Parameters:
`moduleSpecAdv` - the module specification of the server/service to be contacted
Returns:
the reply from the server - null if none is received

---

printServiceList

```
protected void printServiceList(fsgrid.ServiceDescription[]serviceArray)
```

Presents a list of services that this client has found. UI classes should override this method to provide their own way of presenting the information to the user.

Parameters:
`serviceArray` - an array of `ServiceDescription`s that will be presented to the user

---

returnSelection

```
public java.lang.String returnSelection(intvalue)
```

This method is called when the user has made a selection from the list of services. It uses sendMessageToServer to contact the appropriate server.

Parameters:
`value` - the value of the user's selection - should be a number between 1 and the number of services available
Returns:
the reply from the server

---

getCollectedAdvertisements

```
public fsgrid.ServiceDescription[] getCollectedAdvertisements()
```

Returns the service descriptions this client has collected

Returns:
an array of `ServiceDescription`s that this client has collected

Class GridServer

java.lang.Object
  └ fsgrid.Node
    └ fsgrid.Server
      └ **fsgrid.GridServer**

All Implemented Interfaces:
net.jxta.discovery.DiscoveryListener, java.util.EventListener, net.jxta.rendezvous.RendezvousListener, java.lang.Runnable

---

public class **GridServer**
extends Server

A `GridServer` is a Grid-adapted version of a standard `Server`. It can store a GSH (Grid Service Handle), a grid service "address." It also deals with some JXTA settings (configuration directory, user name, password).

Field Summary

Fields inherited from class fsgrid.Node
advThreshold, description, discoveryService, gridDiscoveryService, gridPeerGroup, name

Constructor Summary	
GridServer(java.lang.String name, java.lang.String description, java.lang.String GSH)	
GridServer(java.lang.String name, java.lang.String description, java.lang.String GSH, java.lang.String version, java.lang.String creator)	

Methods inherited from class fsgrid.Server
initializeAdvertisements, run, startConnectionThread, startWork, waitForClientMessages

Methods inherited from class fsgrid.Node
clearAdvCache, discoveryEvent, handleAdvDiscovery, handleGroupDiscovery, handlePeerDiscovery, initializeGridPeerGroup, initializeRendezvous, printMessage, rendezvousEvent, startJxta

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

GridServer

```
public GridServer(java.lang.Stringname,
                  java.lang.Stringdescription,
                  java.lang.StringGSH,
                  java.lang.Stringversion,
                  java.lang.Stringcreator)
```

Parameters:

`name` - the name of the service this server provides
`description` - a description of the service that this server provides
`GSH` - the address of the service this server provides
`version` - the version of the service that this server provides
`creator` - The name of the creator of this service

GridServer

```
public GridServer(java.lang.Stringname,
                  java.lang.Stringdescription,
                  java.lang.StringGSH)
```

Parameters:

`name` - the name of the service this server provides
`description` - a description of the service that this server provides
`GSH` - the address of the service this server provides

Class GridClient

```
java.lang.Object
  └ fsgrid.Node
      └ fsgrid.Client
          └ fsgrid.GridClient
```

All Implemented Interfaces:

net.jxta.discovery.DiscoveryListener, java.util.EventListener,
net.jxta.rendezvous.RendezvousListener

---

public class **GridClient**
extends Client

Javadoc

A `GridClient` is a grid-adapted version of a standard `Client`. The main difference is that it has a `ServiceConnector` member object that it uses for interaction with Grid services. It also deals with some JXTA settings (configuration directory, user name, password).

Field Summary

Fields inherited from class fsgrid.Node
advThreshold, description, discoveryService, gridDiscoveryService, gridPeerGroup, name

Constructor Summary
GridClient(java.lang.String name) A constructor for the GridClient
GridClient(java.lang.String name, java.lang.String description) A constructor for the GridClient
GridClient(java.lang.String name, java.lang.String description, fsgrid.ServiceConnector connector) A constructor for the GridClient

Method Summary	
void	**connectToService**(java.lang.String address) Tries to connect to a service at the given address
void	**noAdvertisementsFound**() Called when no advertisements are found
java.lang.String	**sendMessageToServer**(net.jxta.protocol.ModuleSpecAdvertisement moduleSpecAdv) Reads the address of a grid service from the specURI field, and tries to contact it

Methods inherited from class fsgrid.Client
getCollectedAdvertisements, getServiceAdvertisements, handleAdvDiscovery, handleAdvDiscovery, presentAdvertisements, printServiceList, returnSelection, runDiscovery, runDiscovery, startWork, startWork

Methods inherited from class fsgrid.Node
clearAdvCache, discoveryEvent, handleGroupDiscovery, handlePeerDiscovery, initializeGridPeerGroup, initializeRendezvous, printMessage, rendezvousEvent, startJxta

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

GridClient

```
public GridClient(java.lang.Stringname,
                  java.lang.Stringdescription,
                  fsgrid.ServiceConnectorconnector)
```

A constructor for the GridClient

Parameters:
`name` - the name of the service this client will look for
`description` - a description of the service this client will look for
`connector` - the connector provides the implementation of what the GridTextClient will actually do

---

GridClient

```
public GridClient(java.lang.Stringname,
                  java.lang.Stringdescription)
```

A constructor for the GridClient

Parameters:
`name` - the name of the service this client will look for
`description` - a description of the service this client will look for

GridClient

```
public GridClient(java.lang.Stringname)
```

A constructor for the GridClient

Parameters:
`name` - the name of the service this client will look for

Method Detail

sendMessageToServer

```
public java.lang.String sendMessageToServer
(net.jxta.protocol.ModuleSpecAdvertisementmoduleSpecAdv)
```

Reads the address of a grid service from the specURI field, and tries to contact it

Overrides:
`sendMessageToServer` in class `Client`

Parameters:
`moduleSpecAdv` - a module specification advertisement describing a service
Returns:
always null (the String return type is only used because of inheritance)

---

connectToService

```
public void connectToService(java.lang.Stringaddress)
```

Tries to connect to a service at the given address

Parameters:
`address` - the address of the service to connect to

---

noAdvertisementsFound

```
public void noAdvertisementsFound()
```

Called when no advertisements are found

Overrides:
`noAdvertisementsFound` in class `Client`
