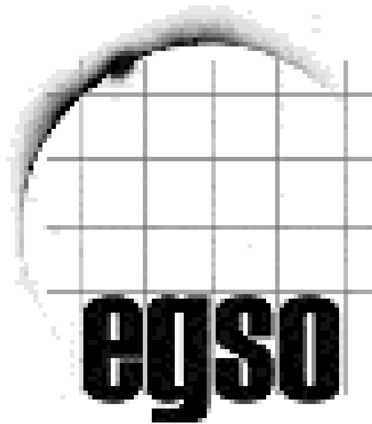


European Grid of Solar Observations

Project n. IST-2001-32409

Title	**Survey on grid and peer-to-peer network technologies**
Document number	EGSO-DE01_01-D02-021001
Date	October 01, 2002
Editor	**Luigi Ciminiera, Andrea Sanna, and Claudio Zunino - Polito**
Contributors	
Distribution	Internal

Table of contents

Table of contents 2
Figure list 4
1. Introduction 5
2. What a computational grid is 5
3. A brief taxonomy of grids 6
4. Some grid technologies 7
4.1 2K: a distributed operating system 7
4.2 AppLeS 7
4.3 Bond 7
4.4 Condor 8
4.5 European DataGrid 8
4.6 Globus 8
4.7 Javelin 8
4.8 Legion 8
4.9 NetSolve 9
4.10 Ninf 9
4.11 Punch 9
5. What a peer-to-peer networking is 9
5.1 Pure peer-to-peer 10
5.2 Peer-to-peer with a Simple Discovery Server 11
5.3 Peer-to-peer with a Discovery and Lookup Server 11
5.4 P2P with Discovery, Lookup, and Content Server 12
6. Analysis of technologies strictly relevant for the EGSO project 12
7. Gnutella 13
7.1 Protocol definition 14
7.2 Descriptor Header 15
7.3 Descriptor Routing 18
7.4 File downloads 19
7.5 Firewalled Servents 19
8. Freenet 19
8.1 Freenet architecture 19
8.2 Network Evolution 22
8.3 Searching 23
8.4 Managing Storage 23
8.5 Performance Analysis 24
9. JXTA 24
9.1 Introduction 24
9.2 Overview of JXTA 24
9.3 JXTA Architecture 25
9.4 JXTA Concepts 26
9.5 Network Architecture 31
9.6 JXTA Protocols 32
9.7 Universal Resource Binding and Rendezvous 33
9.8 Peer Monitoring and Metering 34
9.9 Services 34
10. Globus 34
10.1 GIS: the Globus information service 35
10.2 GRAM: the Globus resource allocation manager 39
10.3 GridFTP 42

11. OGSA 43
11.1 The OGSA Service Model 43
12. Comparison 46
12.1 Resource discovery 46
12.2 Resource management 46
12.3 Resources Description 47
13. Security analysis 47
13.1 Freenet and Gnutella 47
13.2 Globus 47
13.3 JXTA 49
13.4 Security technologies 49
13.5 Summary 50
14. Workflow management systems 51
14.1 Workflow languages 52
15. Distributed data management systems 53
16. References 56

Figure list

Figure 1 Architecture of a computational grid [1]. 6
Figure 2 Architecture of a P2P network [1]. 9
Figure 3 A Pure P2P [3]. 10
Figure 4 P2P with a Simple Discovery Server [3]. 11
Figure 5 P2P with a Discovery, Lookup, and Content Server [3]. 12
Figure 1 Search and retrieval under the Gnutella protocol 14
Figure 2 Ping/Pong Routing 18
Figure 3 Query/QueryHit/Push Routing 19
Figure 4 Freenet request sequence 21
Figure 5 JXTA Software Architecture. 25
Figure 6 Point-to-point and propagate pipes. 29
Figure 7 Request propagation via rendezvous peers. 31
Figure 8 Message routing scenario across a firewall. 32
Figure 9 The Globus structure 35
Figure 10 Architecture overview. 36
Figure 11 Hierarchical discovery. 37
Figure 12 The Globus resource management architecture. 39
Figure 13 Globus resource management architecture. 40
Figure 14 Major components of the GRAM implementation. 42
Figure 15 A possible client-side runtime architecture. 44
Figure 16 Two alternative approaches to the implementation of a service. 45
Figure 17 Architecture of Globus Toolkit (left) and OGSA (right). 46
Figure 18 The use of proxy in the delegation. 48
Figure 19 Trust relationship in Poblano 49

1. Introduction

This document is organized in two different parts: the first one (Section 2 to Section 5) reviews both grid and peer-to-peer network characteristics outlining goals, similarities and differences, while the second one (Section 6 to Section 15) analyzes more in the details technologies considered of great interest in the EGSO program.
Communication is a key element when writing nearly any type of application. An application gains value when it becomes distributed and interacts with other resources available to it on the Internet or intranet.
The most common model for communication over the Internet today is client/server, where there is a client that knows how to request information and post information to a server, and the server knows how to respond to requests from the client.
A grid is a very large scale, generalized distributed network computing system that can scale to Internet size environments with machines distributed across multiple organization and administrative domains implementing client/server paradigm. A brief description of grids has been extracted from [1], while grid taxonomy has been extracted from [2].
A peer-to-peer infrastructure is different from the traditional client/server model because the applications involved act as both clients and servers. A description of the types of peer-to-peer has been extracted from [3]. Figures have been selected from the three above mentioned manuscripts.
Both grids and peer-to-peer (P2P) networks aim to provide users a set of services such as:

- Authentication and security
- A standardized grid-wide name space for files and other resources
- Resource registration and discovery
- Resource accounting
- Resource scheduling
- Job monitoring
- Specialized services

2. What a computational grid is

In a computational grid independent clients can get access to a set of resources through a “transparent” infrastructure (*middleware*) able to route information and implement different services. Therefore, a computational grid is:

a collection of computers, online instruments, data archives, and networks that are connected by a shared set of services that, when taken together, provide users with transparent access to the entire set of resources.

In Figure 1 is shown the typical architecture of a computational grid. The computational grid is analogous to electric power grid. Grid computing allows to couple geographically distributed resources and offers consistent and inexpensive access to resources irrespective of their physical location or access point. It enables sharing, selection, and aggregation of a wide variety of geographically distributed computational resources.
This type of approach is currently used for a large variety of physic, mathematics and biology applications mostly following the Master/slave paradigm.

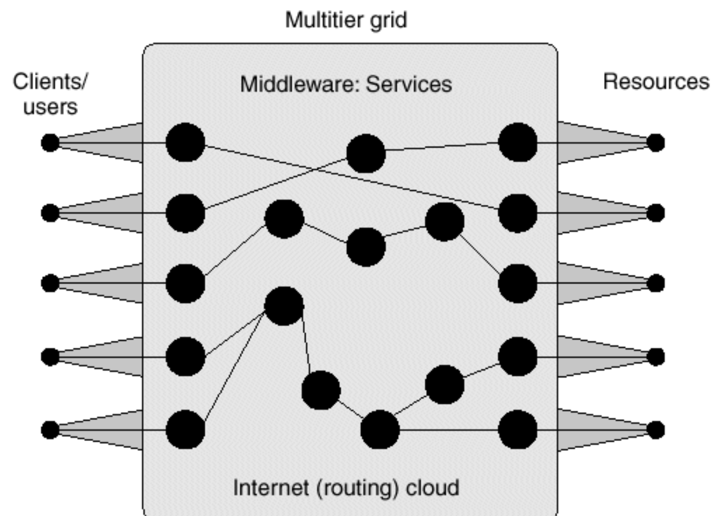


Figure 1 Architecture of a computational grid [1].

3. A brief taxonomy of grids

A first classification of grid systems is based on the kind of application. Grid systems can be divided in:

- **Computational grid.** These systems are devoted to aggregate computational capacity.
- **Data grid.** These systems provides an infrastructure for retrieving data and information from wide area distributed network.
- **Service grid.** These systems provide services that cannot be provided by any single machine

Further, a grid can be categorized observing its internal organization:

- **Flat.** In a flat organization, all machines can be directly communicate with each other without going through an intermediary.
- **Cells.** In a cell structure the machines within the cell communicate between themselves using flat organization. Designated machines within the cell function act as boundary elements that are responsible for all external communications.
- **Hierarchical.** In a hierarchical organization machines in the same level can directly communicate with the machines directly above them or below them

Resources within a grid can be described in two different ways. The first description is a *schema* based approach, where the data that comprises a resource is described via a description language along with some integrity constraints. The second description is a *object model* based approach, where the operations on the resources are defined as part of the resource model. The schema and object approaches are further characterized by the ability to extend themselves (extensible schema/object model).
The resource information store organization determines the cost of implementing the resource management protocols because resource dissemination and discovery may be provided by the data store implementation. It can be:

- **Distributed Objects** data store utilize persistent object services that are provided by language independent object model
- **Network Directory** are based on a database engine and utilize a standard interface language to operate on the schema data.

One of the most important characteristics in a grid system is the Quality of Service (QoS). When a grid job has QoS requirements, may be necessary to negotiate a service level agreement to enforce the desired level of service. Three kind of quality support can be provided:

- **None.** No QoS is supported
- **Soft.** Constraints on QoS cannot be guarantee.
- **Hard.** All nodes in the grid guarantee the QoS requested.

Resource in the grid can be discovered either by an *agent* approach or by a *centralized/distributed* query system.
The scheduling component of the resource management system can be categorized in the following way:

- **Centralized.** There is only one scheduling controller that is responsible for the system-wide decision making.
- **Hierarchical.** In the hierarchical organization, the scheduling controllers are organized in a hierarchy.

Decentralized. This organization does not involve a single scheduler but controllers are not necessarily organized in a hierarchical way.

4. Some grid technologies

4.1 2K: a distributed operating system

The 2K [4] system is a distributed operating system that provides a flexible and adaptable architecture for providing distributed services across a wide variety of platform ranging from personal digital assistant to large scale computers. *2K* adopts a *network-centric* model in which all entities, users, software components, and devices exist in the network and are represented as CORBA objects. Each entity has a network-wide identity, a network-wide profile, and dependencies upon other network entities. When a particular service is instantiated, the entities that constitute that service are assembled. In contrast to existing systems where a large number of non-utilized modules are carried along with the basic system installation, the philosophy is based upon a "What You Need Is What You Get" (WYNIWYG) model. The system configures itself automatically and loads a minimal set of components required for executing the user applications in the most efficient way.

4.2 AppLeS

The AppLeS project [5] primarily focuses on developing scheduling agents for individual applications on production computational grids. AppLeS agents use application and system information to select a viable set of resources. AppLeS uses a services of other resource management systems such as Globus, Legion and NetSolve to execute application tasks.

4.3 Bond

Bond [6] is a Java based distributed object system and agent framework. It implements a message based middleware and associated services like directory, persistence, monitoring and security. Key technical ideas of the Bond Agent framework:

- Multi-plane state machine agent model with multiple semantic engines.
- Component-based architecture (strategies and planes).
- Agent description language (Blueprint and XML).
- Dynamic agent behavior (agent assembly, mobility, surgery, trimming, and lazy loading of strategies).
- Multi-lingual inter-agent communication (KQML and XML).

4.4 Condor

Condor [7] is a high-throughput computing environment that can manage a large collection of diversely owned machines and networks. The Condor environment follows a layered architecture and supports sequential and parallel applications.
By means of its unique remote system call capabilities, Condor preserves a large measure of the originating machine's environment on the execution machine, even if the originating and execution machines do not share a common file system and/or user ID scheme. Condor jobs that consist of a single process are automatically checkpointed and migrated between workstations as needed to ensure eventual completion.

4.5 European DataGrid

The European DataGrid Project [8] focuses on the development of middleware services in order to enable distributed analysis of physics data. The core middleware system is the Globus toolkit with hooks for data Grids. Data on the order of several petabytes will be distributed in a hierarchical fashion to multiple sites worldwide. Global namespaces are required to handle the creation of and access to distributed and replicated data items. Special workload distribution facilities will balance the analysis jobs from several hundred physicists to different places in the Grid in order to have maximum throughput for a large user community. Application monitoring as well as collecting of user access patterns will provide information for access and data distribution optimization.

4.6 Globus

The Globus system [15] enables modular deployment of Grid systems by providing the required basic services and capabilities in the Globus Metacomputing Toolkit (GMT). The toolkit consists of a set of components that implement basic services, such as security, resource location, resource management, data management, resource reservation, and communications. Globus is constructed as a layered architecture in which higher level services can be developed using the lower level core services. Its emphasis is on the hierarchical integration of Grid components and their services.
Globus supports soft QoS via resource reservation The predefined Globus scheduling
policies can be extended by using application level schedulers such as AppLeS and Condor/G. The Globus scheduler in the absence of application level scheduler has a decentralized organization with an ad-hoc extensible scheduling policy.

4.7 Javelin

Javelin [10] is a Java-based infrastructure for *global computing*: Internet-based parallel computing. While **Javelin 2.0** is based on Java *applications,* it retains the Javelin architecture of clients, hosts, and brokers. A *client* is a process seeking computing resources; a *host* is a process offering computing resources; a *broker* is a process that coordinates the allocation of computing resources. Javelin 2.0 supports *piecework* computations: adaptively parallel computations that decompose into a set of sub-computations, each of which is communicationally autonomous, apart from scheduling work and communicating results (e.g., ray tracing and Monte Carlo experiments). It also supports *branch-and-bound* computations. Javelin 2.0 achieves scalability and fault-tolerance by integrating distributed deterministic work-stealing with a distributed deterministic eager scheduler. An additional fault-tolerance mechanism is implemented for replacing hosts that have failed or retreated.

4.8 Legion

Legion [11], an object-based metasystems software project at the University of Virginia, is designed for a system of millions of hosts and trillions of objects tied together with high-speed links. Users working on their home machines see the illusion of a single computer, with access to all kinds of data and physical resources, such as digital libraries, physical simulations, cameras, linear

accelerators, and video streams. Groups of users can construct shared virtual work spaces, to collaborate research and exchange information. This abstraction springs from Legion's transparent scheduling, data management, fault tolerance, site autonomy, and a wide range of security options.

4.9 NetSolve

NetSolve [12] is a client-server system that enables users to solve complex scientific problems remotely. The system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available, and using retry for fault-tolerance solves a problem, and returns the answers to the user. A load-balancing policy is used by the NetSolve system to ensure good performance by enabling the system to use the computational resources available as efficiently as possible.

4.10 Ninf

Ninf [13] is an ongoing global computing infrastructure project which allows users to access computational resources including hardware, software and scientific data distributed across a wide area network with an easy-to-use interface. Users can build applications by calling the libraries with the Ninf Remote Procedure Call, which is designed to provide a programming interface similar to conventional function calls, and is tailored for scientific computation. In order to facilitate location transparency and network-wide parallelism, Ninf metaserver maintains global resource information regarding computational server and databases, allocating and scheduling coarse-grained computation to achieve good global load balancing.

4.11 Punch

PUNCH [14] is a platform for Internet computing that turns the World Wide Web into a distributed computing portal. Users can access and run programs via standard Web browsers. Applications can be installed "as is" in as little as thirty minutes. Machines, data, applications, and other computing services can be located at different sites and managed by different entities.

5. What a peer-to-peer networking is

The difference between a computational grid and a peer-to-peer (P2P) network can be clearly understood comparing Figure 1 and Figure 2. A P2P application is different from the traditional client/server model because the applications involved act as both clients and servers. That is to say, while they are able to request information from other servers, they also have the ability to act as a server and respond to requests for information from other clients at the same time.

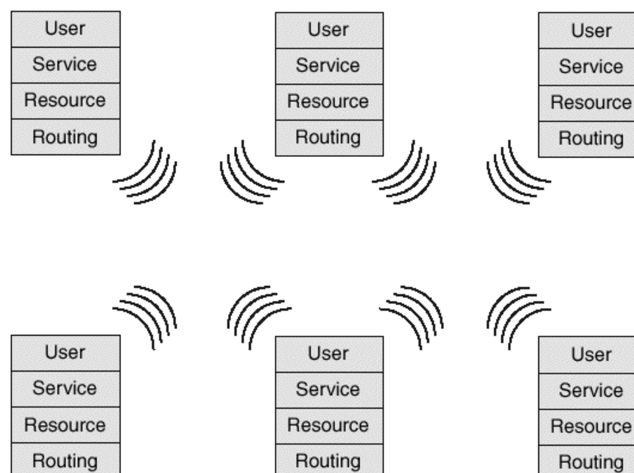


Figure 2 Architecture of a P2P network [1].

Figure 2 shows each node acting as user interface, service provider, message router, and resource repository. Links between such multipurpose nodes tend to be dynamic.

In a peer-to-peer architecture, computers that have traditionally been used solely as clients communicate directly among themselves and can act as both clients and servers, assuming whatever role is most efficient for the network. This reduces the load on servers and allows them to perform specialized services.

A typical peer-to-peer application has the following key features that help define it:

- **Discovering other peers.** The application must be able to find other applications that are willing to share information. Historically, the application finds these peers by registering with a central server that maintains a list of all applications currently willing to share and giving that list to any new applications as they connect to the network. However, there are other means available, such as network broadcasting or discovery algorithms.
- **Querying peers for content.** Once these peers have been discovered, the application can ask them for the content that is desired by the application. Content requests often come from users, but it is highly conceivable that the peer-to-peer application is running on its own and performing its query as a result of some other network request that came to it rather than a specific request made by a user at that machine.
- **Sharing content with other peers.** In the same way that the peer can ask others for content, it can also share content after it has been discovered.

5.1 Pure peer-to-peer

A pure peer-to-peer application has no central server whatsoever, as you can see in Figure 3. It dynamically discovers other peers on the network and interacts with each of them for sending and receiving content. The strength of this type of application is that it does not rely on any one server to be available for registration of its location in order for other peers to find it. At the same time, the lack of a central discovery server poses a problem because a relatively low number of clients can be discovered, thereby limiting the application's reach. In this scenario, a peer can either use information from a local configuration scheme to discover the clients (for example, a configuration entry that tells it who to talk with) or it can employ network broadcasting and discovery techniques such as IP multicast to discover the other peers.

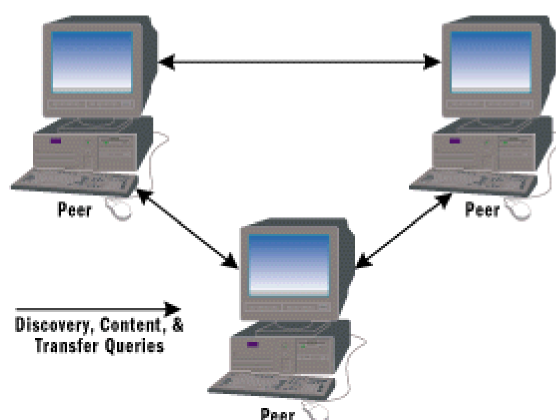


Figure 3 A Pure P2P [3].

Using IP multicast can be problematic since it is not widely deployed on the Internet, but it can be useful in intranet scenarios where the network is more controlled and infrastructure required for multicast is known to exist. Pure peer-to- peer is also being deployed on the Internet in cases where non-multicast schemes are used to discover the peers. In this case, the applications use some other

scheme such as a well-known node approach, where each peer knows about at least one other peer and they share this knowledge with other peers to form a loosely connected mass of nodes.

5.2 Peer-to-peer with a Simple Discovery Server

This architecture, shown in Figure 4, works just like the pure peer-to-peer architecture except that it relies on a central server for discovery of the other peers. In this model, the application usually notifies the central server of its existence at startup time (or login time). The peer application then uses this server to download a list of the other peers participating on its network that it can use to query for content. When content is needed, it goes through the list and contacts each peer individually with its request.

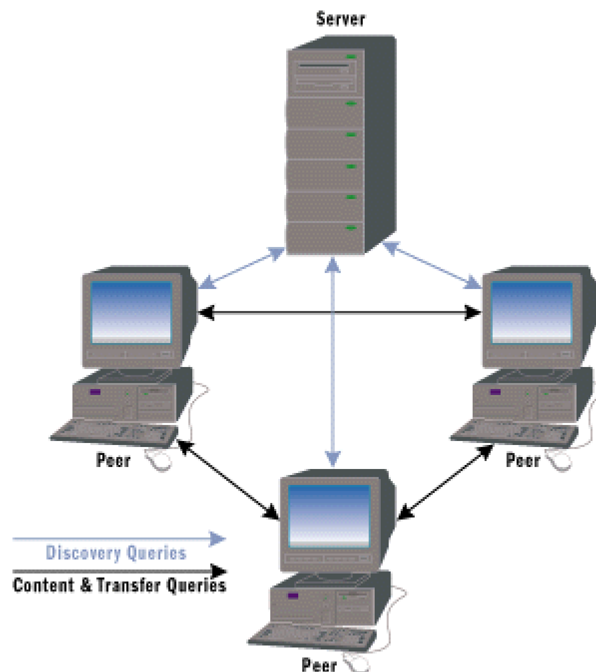


Figure 4 P2P with a Simple Discovery Server [3].

In many cases, it is easier to make this solution scale better than the pure peer-to-peer option because it circumvents the issues of discovery by requiring only one request to the central server. Note that it is possible to make pure peer-to- peer solutions scale extremely well, but if you are able to rely on a server for some of the basic tasks (like discovery), high scalability can be achieved with a lower cost in terms of development time. However, this approach hinges on the availability of the central server. If the central server is not available, the peer-to-peer application will not be able to find other peers.

In addition, requesting content from each individual peer can be quite expensive from a network resource perspective. This may not seem like a big deal if you're thinking about a few peers interacting over a network, but if your app is being written for use over the Internet or a large enterprise environment, this consideration suddenly becomes much more significant, scaling factorially.

5.3 Peer-to-peer with a Discovery and Lookup Server

This model, similar to the one shown in Figure 4, extends the discovery server so that it also includes content lookup services. In this case, the peer application not only registers with a discovery server, but it also uploads a list of its contents at regular intervals. When an application is looking for some particular content, it queries the central server rather than sending a query to each

client. The central server then responds with a list of the clients that contain the requested content, and the peer application can contact those clients directly to retrieve the content.
Quite often this approach will scale better than the previous options because it reduces the number of queries going over the network (arguably one of the scarcest resources). However, this saving will incur a cost on the server. Servers are now more involved in the process of content sharing and the peer's demands will use significant resources.

5.4 P2P with Discovery, Lookup, and Content Server

Just to show that this can actually come full circle, a system can be designed so that the peers can upload the content to the server as well, if you so choose (see Figure 5). This approach effectively becomes the client/server model because the peers are no longer contacting other peers for content. Each peer registers with a server (if needed), queries it for information, and transfers any desired content down from the server. The problem with this approach is that when content is downloaded from all of the clients, the server quickly becomes the bottleneck and is easily overwhelmed by the peers (clients).

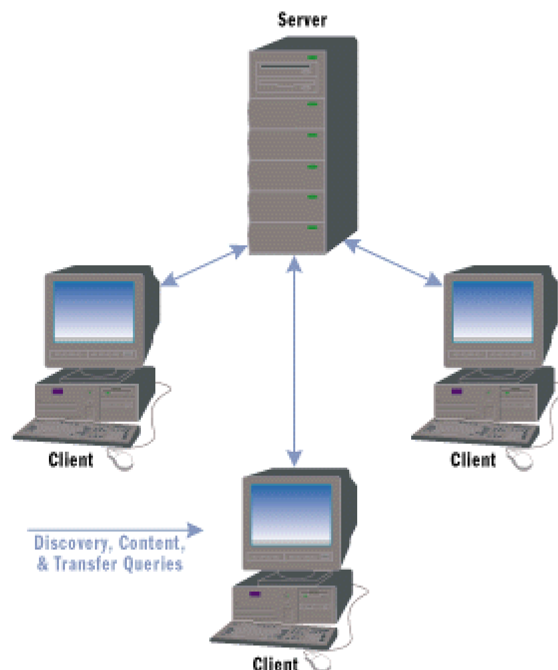


Figure 5 P2P with a Discovery, Lookup, and Content Server [3].

To put this into perspective, consider a peer-to-peer application that shares video content. Let's assume the application supports up to 100,000 peers, each containing megabytes of data. The total amount of content available to any one peer can quickly reach into hundreds of terabytes. While server capacity can always grow, placing such demand on the server can be costly and can create a significant number of bottleneck and reliability issues on the network. Placing all of the demand on the server also means that the substantial resources available on the clients that would be utilized in the peer-to-peer model are potentially wasted in the client/server model.

6. Analysis of technologies strictly relevant for the EGSO project

The second part of this document analyzes some technologies considered of great interest in the EGSO project; in particular, technological issues behind: Freenet [23], Globus [15], Gnutella [56], and JXTA [26] will be reviewed. These four technologies have characteristics suitable to be employed in the EGSO project, in particular, mechanisms of data management, resource discovery, security, and fault tolerance will be analyzed in the details.

Gnutella and Freenet provide a distribute storage space; both act a decentralized architecture in order to accomplish scalability and fault tolerance. In particular, Freenet has shown to be really scalable allowing to effectively interconnect hundred of thousands nodes without affecting network topology. Moreover, Freenet provides security mechanisms both to avoid loss of data and to guarantee the privacy of data owners. Gnutella and Freenet technology will be analyzed in Section 7 and 8, respectively.

JXTA technology, presented in Section 9, is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner. Project JXTA is an open-source project that defines a set of protocols for ad hoc, pervasive, peer-to-peer computing. The Project JXTA protocols establish a virtual network overlay on top of the Internet and non-IP networks, allowing peers to directly interact and organize independently of their network location (firewalls or NATs).

The software toolkit developed within the Globus project comprises a set of components that implement basic services for security, resource location, resource management, data access, communication, and so on. The Open Grid Services Architecture (OGSA), presented more in the details in Section 10, is a proposed evolution of the Globus Toolkit towards a Grid system architecture based on an integration of Grid and Web services concepts and technologies. A basic premise of OGSA is that everything is represented by a *service*: a network enabled entity that provides some capability through the exchange of messages. Computational resources, storage resources, networks, programs, databases, and so forth are all services. This adoption of a uniform service-oriented model means that all components of the environment are virtual. More specifically, OGSA represents everything as a *Grid service*: a Web service that conforms to a set of conventions and supports standard interfaces for such purposes as lifetime management. This core set of consistent interfaces, from which all Grid services are implemented, facilitates the construction of hierarchal, higher-order services that can be treated in a uniform way across layers of abstraction.

Moreover, a set of technologies such as workflow languages and distributed data management systems potentially involved in the EGSO project will be analyzed (Sections 14 and 15).

7. Gnutella

The description of the Gnutella protocol can be found in *The Gnutella Protocol Specification v0.4* (Document Revision 1.2) [18] and in [19].
Gnutella is a protocol for distributed search. Although the Gnutella protocol supports a traditional client/centralized server search paradigm, Gnutella's distinction is its peer-to-peer, decentralized model. In this model, every client is a server, and vice versa. Gnutella nodes are often called *servents.* Due to its distributed nature, a network of servents that implements the Gnutella protocol is highly fault-tolerant, as operation of the network will not be interrupted if a subset of servents goes offline.
Gnutella is a network of equal peers that selectively respond to queries. Each node attempts to establish a number of simultaneous connections to other nodes that constitute its neighbor set.

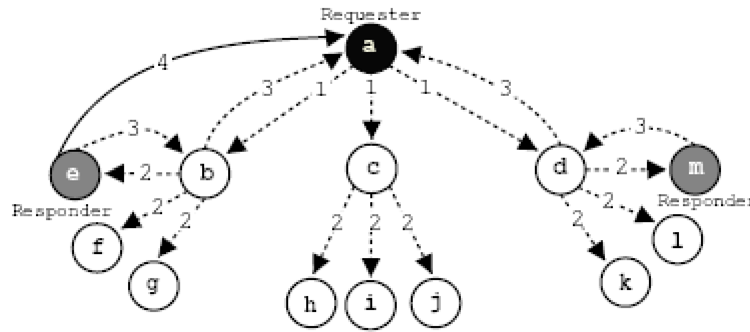


Figure 6 Search and retrieval under the Gnutella protocol [19].

Figure 6 depicts a typical search and retrieval scenario. (1) Node a broadcasts a query q to all nodes in its neighbor set N. (2) Each node in N forwards q to its neighbors. (3) Any node able to satisfy q (in this case nodes e and m) returns via the request chain a brief description of its hits to node a. (4) After reading the descriptions, the user chooses to download one of the hits returned by e (via direct TCP/IP connection). The user is satisfied with this hit and the search/retrieval process is complete. Had the user been unsatisfied, he could retrieve any of the other hits in e's result set or m's result set.

The Gnutella protocol does not specify how nodes must process incoming queries. Thus, it is well-suited for search across heterogeneous data sources. For example, when evaluating the query string "1+1", one node might recognize a mathematical expression and return "2", another might return results from a news database, and so on. To the extent that it is difficult to determine whether an upstream node is forwarding or initiating a query, there is query anonymity. However, due to the direct connection between peers during file transfer, there is no download anonymity. Studies [20,21] indicate that Gnutella's search mechanism does not scale.

Every query reaches an exponentially increasing number of nodes and generates a prohibitive amount of network bandwidth. There has been some work incorporating hub nodes called reflectors or super peers into the network [22]. These super peers are broadband servers that cache incoming queries and prevent them from propagating further. This is a awed solution because it introduces identifiable points of failure into the network, requires a central managing authority, and is vulnerable to the SlashDot effect.

7.1 Protocol definition

The Gnutella protocol defines the way in which servents communicate over the network. It consists of a set of descriptors used for communicating data between servents and a set of rules governing the inter-servent exchange of descriptors. Currently, the following descriptors are defined:

Descriptor	Description
Ping	Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHit	The response to a Query. This descriptor provides the

	recipient with enough information to acquire the data matching the corresponding Query.
Push	A mechanism that allows a firewalled servent to contribute file-based data to the network.

A Gnutella servent connects itself to the network by establishing a connection with another servent currently on the network. The acquisition of another servent's address is not part of the protocol definition. Once the address of another servent on the network is obtained, a TCP/IP connection to the servent is created, and a Gnutella connection request string (ASCII encoded) may be sent.
A servent may reject an incoming connection request for a variety of reasons - a servent's pool of incoming connection slots may be exhausted, or it may not support the same version of the protocol as the requesting servent, for example. Once a servent has connected successfully to the network, it communicates with other servents by sending and receiving Gnutella protocol descriptors. Each descriptor is preceded by a Descriptor Header.

7.2 Descriptor Header

A Descriptor Headers have the following structure:

Descriptor ID

A 16-byte string uniquely identifying the descriptor on the network

Payload Descriptor

0x00 = Ping
0x01 = Pong
0x40 = Push
0x80 = Query
0x81 = QueryHit

TTL

Time To Live. The number of times the descriptor will be forwarded by Gnutella servents before it is removed from the network. Each servent will decrement the TTL before passing it on to another servent. When the TTL reaches 0, the descriptor will no longer be forwarded.

Hops

The number of times the descriptor has been forwarded. As a descriptor is passed from servent to servent, the TTL and Hops fields of the header must satisfy the following condition: TTL(0) = TTL(i) + Hops(i). Where TTL(i) and Hops(i) are the value of the TTL and Hops fields of the header at the descriptor's i-th hop, for $i >= 0$.

Payload Length

The length of the descriptor immediately following this header. The next descriptor header is located exactly Payload_Length bytes from the end of this header i.e. there are no gaps or pad bytes in the Gnutella data stream.

The TTL is the only mechanism for expiring descriptors on the network. Abuse of the TTL field will lead to an unnecessary amount of network traffic and poor network performance.
The Payload Length field is the only reliable way for a servent to find the beginning of the next descriptor in the input stream. Therefore, servents should rigorously validate the Payload Length field for each descriptor received.
Immediately following the descriptor header, is a payload consisting of one of the following descriptors:

Ping (0x00)
Ping descriptors have no associated payload and are of zero length. A Ping is simply represented by a Descriptor Header whose Payload_Descriptor field is 0x00 and whose Payload_ Length field is 0x00000000.
A servent uses Ping descriptors to actively probe the network for other servents. A servent receiving a Ping descriptor may elect to respond with a Pong descriptor, which contains the address of an active Gnutella servent (possibly the one sending the Pong descriptor) and the amount of data it's sharing on the network. This specification makes no recommendations as to the frequency at which a servent should send Ping descriptors, although servent implementers should make every attempt to minimize Ping traffic on the network.

Pong (0x01)
Pong Descriptors have the following structure [18]:

Port *The port number on which the responding host can accept incoming connections.*

IP Address *The IP address of the responding host (This field is in big-endian format).*

Number of Files Shared
The number of files that the servent with the given IP address and port is sharing on the network.

Number of Kilobytes Shared
The number of kilobytes of data that the servent with the given IP address and port is sharing on the network. Pong descriptors are only sent in response to an incoming Ping descriptor. It is valid for more than one Pong descriptor to be sent in response to a single Ping descriptor. This enables host caches to send cached servent address information in response to a Ping request.

Query (0x80)
Query Descriptors have the following structure [18]:

Minimum Speed
The minimum speed (in kb/second) of servents that should respond to this message. A servent receiving a Query descriptor with a Minimum Speed field of n kb/s should only respond with a QueryHit if it is able to communicate at a speed >= n kb/s

Search Criteria
A nul (i.e. 0x00) terminated search string. The maximum length of this string is bounded by the Payload_Length field of the descriptor header.

QueryHit (0x81)
QueryHit Descriptors have the following structure [18]:

Number of Hits

The number of query hits in the result set (see below).

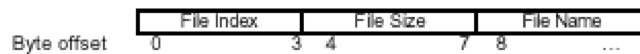
Port

The port number on which the responding host can accept incoming connections.

IP Address *The IP address of the responding host (This field is in big-endian format).*

Speed *The speed (in kb/second) of the responding host.*

Result Set *A set of responses to the corresponding Query. This set contains Number_of_Hits elements, each with the following structure* [18]*:*



File Index

A number, assigned by the responding host, which is used to uniquely identify the file matching the corresponding query.

File Size

The size (in bytes) of the file whose index is File_Index.

File Name

The double-nul (i.e. 0x0000) terminated name of the filewhose index is File_Index.

The size of the result set is bounded by the size of the Payload_Length field in the Descriptor Header.

Servent Identifier

A 16-byte string uniquely identifying the responding servent on the network. This is typically some function of the servent's network address. The Servent Identifier is instrumental in the operation of the Push Descriptor (see below).

QueryHit descriptors are only sent in response to an incoming Query descriptor. A servent should only reply to a Query with a QueryHit if it contains data that strictly meets the Query Search Criteria. The Descriptor_Id field in the Descriptor Header of the QueryHit should contain the same value as that of the associated Query descriptor. This allows a servent to identify the QueryHit descriptors associated with Query descriptors it generated.

Push (0x40)

Push Descriptors have the following structure [18]:

Servent Identifier

The 16-byte string uniquely identifying the servent on thenetwork who is being requested to push the file with index File_Index. The servent initiating the push request should set this field to the Servent_Identifier returned in the corresponding QueryHit descriptor. This allows the recipient of a push request to determine whether or not it is the target of that request.

File Index

The index uniquely identifying the file to be pushed from the target servent. The servent initiating the push request should set this field to the value of one

of the File_Index fields from the Result Set in the corresponding QueryHit descriptor.

IP Address

The IP address of the host to which the file with File_Index should be pushed.(This field is in big-endian format).

Port

The port to which the file with index File_Index should be pushed.

7.3 Descriptor Routing

The peer-to-peer nature of the Gnutella network requires servents to route network traffic (queries, query replies, push requests, etc.) appropriately. A well-behaved Gnutella servent will route protocol descriptors according to the following rules:

- Pong descriptors may only be sent along the same path that carried the incoming Ping descriptor. This ensures that only those servents that routed the Ping descriptor will see the Pong descriptor in response. A servent that receives a Pong descriptor with Descriptor ID = n, but has not seen a Ping descriptor with Descriptor ID = *n* should remove the Pong descriptor from the network.
- QueryHit descriptors may only be sent along the same path that carried the incoming Query descriptor. This ensures that only those servents that routed the Query descriptor will see the QueryHit descriptor in response. A servent that receives a QueryHit descriptor with Descriptor ID = n, but has not seen a Query descriptor with Descriptor ID = *n* should remove the QueryHit descriptor from the network.
- Push descriptors may only be sent along the same path that carried the incoming QueryHit descriptor. This ensures that only those servents that routed the QueryHit descriptor will see the Push descriptor. A servent that receives a Push descriptor with Servent_Identifier = n, but has not seen a QueryHit descriptor with Servent Identifier = *n* should remove the Push descriptor from the network. Push descriptors are routed by Servent_Identifier, not by Descriptor_Id.
- A servent will forward incoming Ping and Query descriptors to all of its directly connected servents, except the one that delivered the incoming Ping or Query.
- A servent will decrement a descriptor header's TTL field, and increment its Hops field, before it forwards the descriptor to any directly connected servent. If, after decrementing the header's TTL field, the TTL field is found to be zero, the descriptor is not forwarded along any connection.
- A servent receiving a descriptor with the same Payload Descriptor and Descriptor ID as one it has received before, should attempt to avoid forwarding the descriptor to any connected servent. Its intended recipients have already received such a descriptor, and sending it again merely wastes network bandwidth. Examples of Ping/Pong Routing and Query/QueryHit/Push Routing are shown in Figure 7 and Figure 8 [18].

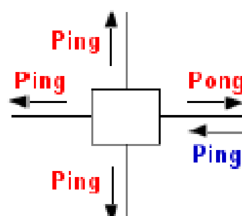


Figure 7 Ping/Pong Routing

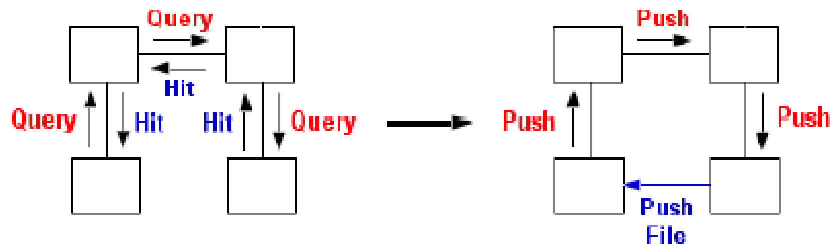


Figure 8 Query/QueryHit/Push Routing

7.4 File downloads

Once a servent receives a QueryHit descriptor, it may initiate the direct download of one of the files described by the descriptor's Result Set. Files are downloaded out-of-network i.e. a direct connection between the source and target servent is established in order to perform the data transfer. File data is never transferred over the Gnutella network. The file download protocol is HTTP.

7.5 Firewalled Servents

It is not always possible to establish a direct connection to a Gnutella servent in an attempt to initiate a file download. The servent may, for example, be behind a firewall that does not permit incoming connections to its Gnutella port. If a direct connection cannot be established, the servent attempting the file download may request that the servent sharing the file "push" the file instead. A servent can request a file push by routing a Push request back to the servent that sent the QueryHit descriptor describing the target file. The servent that is the target of the Push request (identified by the *Servent Identifier* field of the Push descriptor) should, upon receipt of the Push descriptor, attempt to establish a new TCP/IP connection to the requesting servent (identified by the *IP Address* and *Port* fields of the Push descriptor). If this direct connection cannot be established, then it is likely that the servent that issued the Push request is itself behind a firewall. In this case, file transfer cannot take place.

8. Freenet

Freenet [23,24,25], is a distributed information storage system designed to address information privacy and survivability concerns. A beta version of the software is currently available under open source at http://www.freenetproject.org/.
In simulations of up to 200,000 nodes, executed by the developers, Freenet has proved scalable and fault tolerant. It operates as a self-organizing P2P network that pools unused disk space across potentially hundreds of thousands of desktop computers to create a collaborative virtual file system. To increase network robustness and eliminate single points of failure, Freenet employs a completely decentralized architecture.

8.1 Freenet architecture

Freenet participants each run a node that provides the network some storage space. To add a new file, a user sends the network an insert message containing the file and its assigned location-independent globally unique identifier (GUID), which causes the file to be stored on some set of nodes. During a file's lifetime, it might migrate to or be replicated on other nodes. To retrieve a file, a user sends out a request message containing the GUID key. When the request reaches one of the nodes where the file is stored, that node passes the data back to the request's originator.

GUID Keys
Freenet GUID keys are calculated using SHA-1 secure hashes. The network employs two main types of keys: *content-hash key*s, used for primary data storage, and *signed-subspace key*s, intended for higher-level human use. The two are analogous to inodes and filenames in a conventional file system.

Content-hash keys
The content-hash key (CHK) is the low-level data-storage key and is generated by hashing the contents of the file to be stored. This process gives every file an unique absolute identifier (SHA-1 collisions are considered nearly impossible) that can be verified quickly. Unlike with URLs, you can be certain that a CHK reference will point to the exact file intended. CHKs also permit identical copies of a file inserted by different people to be automatically coalesced because every user will calculate the same key for the file.

Signed-subspace keys
The signed-subspace key (SSK) sets up a personal namespace that anyone can read but only its owner can write to. Signing the file with the private half of the key provides an integrity check as every node that handles a signed-subspace file verifies its signature before accepting it. To retrieve a file from a subspace, only the subspace's public key is needed and the descriptive string, from which the SSK can be recreated. Adding or updating a file, on the other hand, requires the private key in order to generate a valid signature.
Typically, SSKs are used to store indirect files containing pointers to CHKs rather than to store data files directly. Indirect files combine the readability and publisher authentication of SSKs with the fast verification of CHKs. They also allow data to be updated while preserving referential integrity. To perform an update, the data's owner first inserts a new version of the data, which will get a new CHK because the file contents are different.
The owner then updates the SSK to point to the new version. The new version will be available by the original SSK, and the old version will remain accessible by the old CHK. Indirect files can also be used to split large files into multiple pieces by inserting each part under a separate CHK and creating an indirect file that points to all the parts.
Finally, indirect files can be used to create hierarchical namespaces from directory files that point to other files and directories. SSKs can also be used to implement an alternative domain name system for nodes that change address frequently. Each such node would have its own subspace, and can be contacted by looking up its public to retrieve the current address.

Messaging and Privacy
Freenet was designed from the beginning under the assumption of hostile attack from both inside and out. Therefore, it intentionally makes it difficult for nodes to direct data toward themselves and keeps its routing topology dynamic and concealed.
Privacy in Freenet is maintained using a variation of Chaum's mix-net scheme for anonymous communication. Rather than move directly from sender to recipient, messages travel through node-to-node chains, in which each link is individually encrypted, until the message finally reaches its recipient. Because each node in the chain knows only about its immediate neighbors, the end points could be anywhere among the network's hundreds of thousands of nodes, which are continually exchanging indecipherable messages. Not even the node immediately after the sender can tell whether its predecessor was the message's originator or was merely forwarding a message from another node. Similarly, the node immediately before the receiver can't tell whether its successor is the true recipient or will continue to forward it. This arrangement is intended to protect not only information producers and consumers (at the beginning of chains), but also information holders (at

the end of chains). Protecting the latter prevents an adversary from destroying a file by attacking all of its holders.

Routing

Routing queries to data is the most important element of the Freenet system. The simplest routing method, used by services like Napster, is to maintain a central index of files, so that users can send requests directly to information holders. Unfortunately, centralization creates a single point of failure that is easy to attack

On the other hand, systems like Gnutella broadcast queries to every connected node within some radius. Although this process would eventually find the answer, it is clearly wasteful and unscalable.

Freenet avoids both problems by using a steepest-ascent hill-climbing search: each node forwards queries to the node that it thinks is closest to the target.

Requesting files

Every node maintains a routing table that lists the addresses of other nodes and the GUID keys it thinks they hold. When a node receives a query, it first checks its own store, and if it finds the file, returns it with a tag identifying itself as the data holder. Otherwise, the node forwards the request to the node in its table with the closest key to the one requested. That node then checks its store, and so on. If the request is successful, each node in the chain passes the file back upstream and creates a new entry in its routing table associating the data holder with the requested key. Depending on its distance from the holder, each node might also cache a copy locally. To conceal the identity of the data holder, nodes will occasionally alter reply messages, setting the holder tags to point to themselves before passing them back up the chain. Later requests will still locate the data because the node retains the true data holder's identity in its own routing table and forwards queries to the correct holder. Routing tables are never revealed to other nodes. To limit resource usage, the requester gives each query a time-to-live limit that is decremented at each node. If the TTL expires, the query fails, although the user can try again with a higher TTL (up to some maximum). Because the TTL can give clues about where in the chain the requester is, Freenet offers the option of enhancing security by adding an initial mix-net route before normal routing. This effectively repositions the start of the chain away from the requester. If a node sends a query to a recipient that is already in the chain, the message is bounced back and the node tries to use the next-closest key instead. If a node runs out of candidates to try, it reports failure back to its predecessor in the chain, which then tries its second choice, and so on.

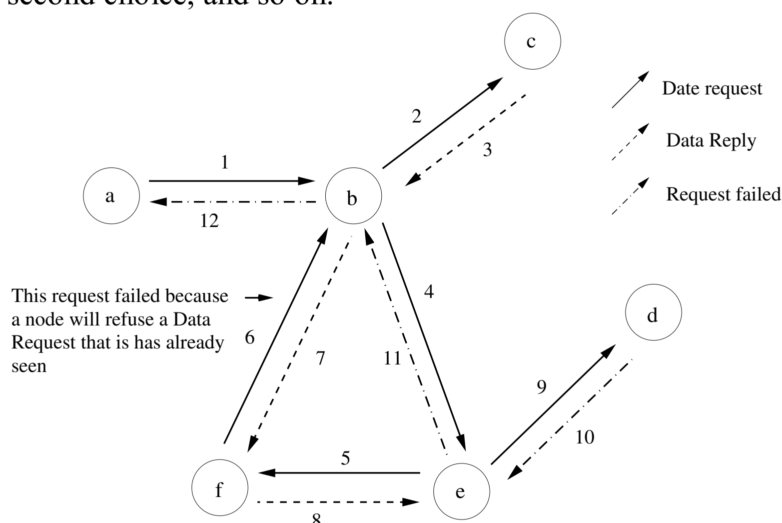


Figure 9 Freenet request sequence [24].

Figure 9 depicts a typical request sequence. The user initiates a request at node *A* and forwards the request to B, which forwards it to C. Node *C* is unable to contact any other nodes and returns a "request failed" message to B. Node *B* then tries its second choice, E, which forwards the request to F. Node *F* forwards the request to B, which detects a loop and bounces the message back. Unable to contact any additional nodes, node *F* backtracks one step to E, which forwards the request to its second choice, D, and locates the file. *D* returns the file via *E* and *B* back to A, which sends it to the user. Along the way, E, B, and *A* might also cache the file. With this approach, the request homes in closer with each hop until the key is found. A subsequent query for this key will tend to approach the first request's path, and a locally cached copy can satisfy the query after the two paths converge. Subsequent queries for similar keys will also jump over intermediate nodes to one that has previously supplied similar data. Nodes that reliably answer queries will be added to more routing tables, and hence, will be contacted more often than nodes that do not.

Inserting files

An insert message follows the same path that a request for the same key would take, sets the routing table entries in the same way, and stores the file on the same nodes. Thus, new files are placed where queries would look for them. To insert a file, a user assigns it a GUID key and sends an insert message to the user's own node containing the new key with a TTL value that represents the number of copies to store. Upon receiving an insert, a node checks its data store to see if the key already exists. If so, the insert fails — either because the file is already in the network (for CHKs) or the user has already inserted another file with the same description (for SSKs). In the latter case, the user should choose a different description or perform an update rather than an insert. (At the moment the update mechanism is not yet implemented.) If the key does not already exist in the node's data store, the node looks up the closest key and forwards the message to the corresponding node as it would for a query. If the TTL expires without collision, the final node returns an "all clear" message. The user then sends the data down the path established by the initial insert message. Each node along the path verifies the data against its GUID, stores it, and creates a routing table entry that lists the data holder as the final node in this chain. As with requests, if the insert encounters a loop or a dead end, it backtracks to the second-nearest key, then the third-nearest, and so on, until it succeeds.

Data Encryption

For political or legal reasons, node operators might wish to remain ignorant of the contents of their data stores. To this end, data encryption before insertion is encouraged. The network proper knows nothing about this level of encryption because it just ships already encrypted bits. Data encryption keys are not used in routing or included in network messages. Inserters distribute them directly to end users at the same time as the corresponding GUIDs. Thus, node operators cannot read their own files, but users can decrypt them after retrieval. Node operators cannot gain any information by looking at GUIDs, either, because the hashes used to generate them scramble any identifying characteristics. From a node operator's point of view, the data store consists only of random GUIDs attached to opaque data.

8.2 Network Evolution

The network evolves over time as new nodes join and existing nodes create new connections after handling queries. As more requests are handled, local knowledge about other nodes in the network improves, and routes adapt to become more accurate without requiring global directories.

Adding Nodes

To join the network, a new node first generates a public-private key pair for itself. This pair serves to logically identify the node and is used to sign a physical address reference. Next, the node sends

an announcement message including the public key and physical address to an existing node, located through some out-of-band means such as personal communication or lists of nodes posted on the Web, with a user-specified TTL. The receiving node notes the new node's identifying information and forwards the announcement to another node chosen randomly from its routing table. The announcement continues to propagate until its TTL runs out. At that point, the nodes in the chain collectively assign the new node a random GUID in the key-space using a cryptographic protocol for shared random number generation that prevents any participant from biasing the result. This procedure assigns the new node responsibility for a region of keyspace that all participants agree on while guaranteeing that a malicious node cannot influence the assignment for a specific key that it might want to attack.

Training Routes
As more requests are processed, the network's routing should become better trained. Nodes' routing tables should specialize in handling clusters of similar keys because each node will mostly receive requests for keys that are similar to the keys it is associated with in other nodes' routing tables. When those requests succeed, the node learns about previously unknown nodes that can supply such keys and creates new routing entries for them. As the node gains more experience in handling queries for those keys, it will successfully answer them more often and, in a positive feedback loop, get asked about them more often. Nodes' data stores should also specialize in storing clusters of files with similar keys. Because inserts follow the same paths as requests, similar keys tend to cluster in the nodes along those paths. Nodes should similarly cluster files cached after requests because most requests will be for similar keys. Taken together, the twin effects of clustering in routing tables and data stores should improve
the effectiveness of future queries in a self-reinforcing cycle.

Key Clustering
Because GUID keys are derived from hashes, the closeness of keys in a data store is unrelated to the corresponding files' contents. This lack of semantic closeness is unimportant, however, because the routing algorithm is based on the locations of particular keys, rather than particular topics. In fact, hashes are useful because they ensure that similar works will be scattered throughout the network, lessening the chances that a single node's failure will make an entire category of files unavailable. Similarly, the contents of any given subspace will be scattered across different nodes, which increases robustness.

8.3 Searching

One open issue is how users can search the network for relevant keys. This is similar to the problem of searching the Web, and similar solutions are possible: Freenet can be spidered, or individuals can publish lists of bookmarks. However, these approaches are not entirely satisfactory in terms of Freenet's design goals. One simple approach for a true Freenet search would be to create a special public subspace for indirect keyword files. When authors insert files, they could also insert several indirect files corresponding to search keywords for the original file. The system would allow multiple keyword files with the same key to coexist (unlike with normal files), and requests for such keys could return multiple matches. Managing a large number of indirect files for common keywords would be difficult, however, because all the files with the same name would be attracted to the same nodes. A more sophisticated approach might use some type of distributed search over detailed metadata descriptors inserted along with the original files.

8.4 Managing Storage

Given finite disk space, the system must sometimes decide which files to keep. It currently prioritizes space allocation by popularity, as measured by the frequency of requests per file. Each node orders the files in its data store by time of last request, and when a new file arrives that cannot

fit in the space available, the node deletes the least recently requested files until there is room. Because routing table entries are smaller, they can be kept around longer than files. Evicted files don't necessarily disappear right away because the node can respond to a later request for the file using its routing table to contact the original data holder, which might be able to supply another copy. Freenet's data holder pointers have a treelike structure. Nodes at the leaves might see only a few local requests for a file, but those higher up the tree receive requests from a larger part of the network, which makes their copies more popular. File distribution is therefore determined by two competing forces: *tree growth* and *prunin*g. The query-routing mechanism automatically creates more copies in an area of the network where a file is requested, and the tree grows in that direction. This improves response time and prevents overloading when the popularity of a file increases suddenly. Files that go unrequested in another part of the network are subject to deletion. As that part of the tree shrinks, space is freed up for other files. The net effect is that the number and location of copies adjust to the demand for each file.

8.5 Performance Analysis

To test Freenet's scalability, a simulated network of 20 nodes has been created; nodes have been initially connected in a ring topology. Randomly generated inserts of files have been sent in the network, interspersed with random requests for files that had already been inserted (all with TTL = 20). After every five inserts and requests, a new node was created, which announced itself to a random existing node with TTL = 10. Network's performance has been measured after every hundred inserts and requests by issuing a set of test requests for previously inserted files and recording the resulting path length distribution (the number of hops actually required to find the data). This continued until the network reached 200,000 nodes. Simulations showed that the network can locate files quickly with a median path length of just 8 hops in a 10,000 node network.

9. JXTA

9.1 Introduction

This survey reviews the JXTA[25] technology: is a network programming and computing platform that is designed to solve a number of problems in modern distributed computing, especially in the area broadly referred to as peer-to-peer computing.
Project JXTA is an open-source project that defines a set of protocols for ad hoc, pervasive, peer-to-peer computing. The Project JXTA protocols establish a virtual network overlay on top of the Internet and non-IP networks, allowing peers to directly interact and organize independently of their network location (firewalls or NATs).
Description of JXTA has been extracted from [26,27,28].

9.2 Overview of JXTA

Project JXTA has a set of objectives that are derived from what it has perceived as shortcoming of many peer-to-peer systems in existence or under development.

- *Interoperability.* JXTA technology is designed to enable interconnected peers to easily locate each other, communicate with each other, participate in community-based activities, and offer services to each other seamlessly across different P2P systems and different communities. Many peer-to-peer systems are built for delivering a single type of services. For example, Napster provides music file sharing, Gnutella provides generic file sharing, and AIM provides instant messaging. Given the diverse characteristics of these services and the lack of a common underlying P2P infrastructure, each P2P software vendor tends to create incompatible systems — none of them able to interoperate with one another. This means each vendor creates its own P2P user community, duplicating efforts in creating

software and system primitives commonly used by all P2P systems. Moreover, for a peer to participate in multiple communities organized by different P2P implementations, the peer must support multiple implementations, each for a distinct P2P system or community, and serve as the aggregation point.

- *Platform independence.* JXTA technology is designed to be independent of programming languages (such as C or the Java™ programming language), system platforms (such as the Microsoft Windows and UNIX ® operating systems), and networking platforms (such as TCP/IP or Bluetooth).
- *Ubiquity*. JXTA technology is designed to be implementable on every device with a digital heartbeat, including sensors, consumer electronics, PDAs, appliances, network routers, desktop computers, data-center servers, and storage systems.

Project JXTA envisions a world where each peer, independent of software and hardware platform, can benefit and profit from being connected to millions of other peers.

9.3 JXTA Architecture

The Project JXTA software architecture is divided into three layers, as shown in Figure 10.

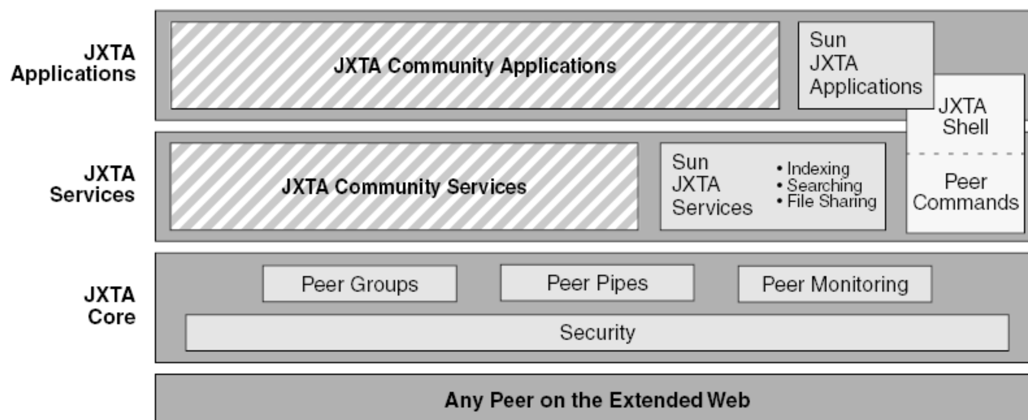


Figure 10 JXTA Software Architecture [28].

- *Platform Layer (JXTA Core)*. The platform layer, also known as the JXTA core, encapsulates minimal and essential primitives that are common to P2P networking. It includes building blocks to enable key mechanisms for P2P applications, including discovery, transport (including firewall handling), the creation of peers and peer groups, and associated security primitives.
- *Services Layer.* The services layer includes network services that may not be absolutely necessary for a P2P network to operate, but are common or desirable in the P2P environment. Examples of network services include searching and indexing, directory, storage systems, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, and PKI (Public Key Infrastructure) services.
- *Applications Layer.* The applications layer includes implementation of integrated applications, such as P2P instant messaging, document and resource sharing, entertainment content management and delivery, P2P Email systems, distributed auction systems, and many others.

The boundary between services and applications is not rigid. An application to one customer can be viewed as a service to another customer. The entire system is designed to be modular, allowing developers to pick and choose a collection of services and applications that suits their needs.

The JXTA network consists of a series of interconnected nodes, or *peers*. Peers can self-organize into *peer groups*, which provide a common set of services. Examples of services that could be provided by a peer group include document sharing or chat applications.
JXTA peers advertise their services in XML documents called *advertisements.* Advertisements enable other peers on the network to learn how to connect to, and interact with, a peer's services.
JXTA peers use *pipes* to send *messages* to one another. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Messages are simple XML documents whose envelope contains routing, digest, and credential information. Pipes are bound to specific *endpoints*, such as a TCP port and associated IP address.
These concepts are described in detail in the following sections.

Three essential aspects of the JXTA architecture distinguish it from other distributed network models:

- The use of XML documents (advertisements) to describe network resources.
- Abstraction of pipes to peers, and peers to endpoints without reliance upon a central naming/addressing authority such as DNS.
- A uniform peer addressing scheme (peer IDs).

9.4 JXTA Concepts

Peers

A *peer* is any networked device that implements one or more of the JXTA protocols. Peers can include sensors, phones, and PDAs, as well as PCs, servers, and supercomputers. Each peer operates independently and asynchronously from all other peers, and is uniquely identified by a Peer ID.
Peers publish one or more network interfaces for use with the JXTA protocols. Each published interface is advertised
as a *peer endpoint,* which uniquely identifies the network interface. Peer endpoints are used by peers to establish direct point-to-point connections between two peers.
Peers are not required to have direct point-to-point network connections between themselves. Intermediary peers may be used to route messages to peers that are separated due to physical network connections or network configuration (e.g., NATS, firewalls, proxies).
Peers spontaneously discover each other on the network to form transient or persistent relationships called peer
groups.

Peer Groups

A *peer group* is a collection of peers that have agreed upon a common set of services. Peers self-organize into peer groups, each identified by a unique peer group ID. Each peer group can establish its own membership policy from open (anybody can join) to highly secure and protected (sufficient credentials are required to join).
Peers may belong to more than one peer group simultaneously. By default, the first group that is instantiated is the Net Peer Group. All peers belong to the Net Peer Group. Peers may elect to join additional peer groups.
The JXTA protocols describe how peers may publish, discover, join, and monitor peer groups; they do not dictate when or why peer groups are created.

There are several motivations for creating peer groups:

- *To create a secure environment* Groups create a local domain of control in which a specific security policy can be enforced. The security policy may be as simple as a plain text

username/password exchange, or as sophisticated as public key cryptography. Peer group boundaries permit member peers to access and publish protected contents. Peer groups form logical regions whose boundaries limit access to the peer group resources.

- *To create a scoping environment* Groups allow the establishment of a local domain of specialization. For example, peers may group together to implement a document sharing network or a CPU sharing network. Peer groups serve to subdivide the network into abstract regions providing an implicit scoping mechanism. Peer group boundaries define the search scope when searching for a group's content.
- *To create a monitoring environment* Peer groups permit peers to monitor a set of peers for any special purpose (e.g., heartbeat, traffic introspection, or accountability).

Groups also form a hierarchical parent-child relationship, in which each group has single parent. Search requests are propagated within the group. The advertisement for the group is published in the parent group in addition to the group itself.

A peer group provides a set of services called *peer group services.* JXTA defines a core set of peer group services. Additional services can be developed for delivering specific services. In order for two peers to interact via a service, they must both be part of the same peer group.

The core peer group services include the following:

- *Discovery Service* The discovery service is used by peer members to search for peer group resources, such as peers, peer groups, pipes and services.
- *Membership Service* The membership service is used by current members to reject or accept new group membership application. Peers wishing to join a peer group must first locate a current member, and then request to join. The application to join is either rejected or accepted by the collective set of current members. The membership service may enforce a vote of peers or elect a designated group representative to accept or reject new membership applications.
- *Access Service* The access service is used to validate requests made by one peer to another. The peer receiving the request provides the requesting peers credentials and information about the request being made to determine if the access is permitted. [Note: not all actions within the peer group need to be checked with the access service; only those actions which are limited to some peers need to be checked.]
- *Pipe Service* The pipe service is used to create and manage pipe connections between the peer group members.
- *Resolver Service* The resolver service is used to send generic query requests to other peers. Peers can define and exchange queries to find any information that may be needed (e.g., the status of a service or the state of a pipe endpoint).
- *Monitoring Service* The monitoring service is used to allow one peer to monitor other members of the same peer group.

Not all the above services must be implemented by every peer group. A peer group is free to implement only the services it finds useful, and rely on the default net peer group to provide generic implementations of non-critical core services.

Network Services

Peers cooperate and communicate to publish, discover, and invoke *network services*. Peers can publish multiple services.

Peers discover network services via the Peer Discovery Protocol.

The JXTA protocols recognize two levels of network services:

- *Peer Services* A peer service is accessible only on the peer that is publishing that service. If that peer should fail, the service also fails. Multiple instances of the service can be run on different peers, but each instance publishes its own advertisement.
- *Peer Group Services* A peer group service is composed of a collection of instances (potentially cooperating with each other) of the service running on multiple members of the peer group. If any one peer fails, the collective peer group service is not affected (assuming the service is still available from another peer member). Peer group services are published as part of the peer group advertisement.

Services can be either pre-installed onto a peer or loaded from the network. In order to actually run a service, a peer may have to locate an implementation suitable for the peer's runtime environment. The process of finding, downloading, and installing a service from the network is similar to performing a search on the Internet for a Web page, retrieving the page, and then installing the required plug-in.

Modules

JXTA modules are an abstraction used to represent any piece of "code" used to implement a behavior in the JXTA world. Network services are the most common example of behavior that can be instantiated on a peer. The module abstraction does not specify what this "code" is: it can be a Java class, a Java jar, a dynamic library DLL, a set of XML messages, or a script. The implementation of the module behavior is left to module implementors. For instance, modules can be used to represent different implementations of a network service on different platforms, such as the Java platform, Microsoft Windows, or the Solaris Operating Environment.

Modules are used by peer groups services, and can also be used by stand-alone services. JXTA services can use the module abstraction to identify the existence of the service (its Module Class), the specification of the service (its Module Specification), or an implementation of the service (a Module Implementation). Each of these components has an associated advertisement, which can be published and discovered by other JXTA peers.

Pipes

JXTA peers use *pipes* to send messages to one another. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Pipes are indiscriminate; they support the transfer of any object, including binary code, data strings, and Java technology-based objects.

The pipe endpoints are referred to as the *input pipe* (the receiving end) and the *output pipe* (the sending end). Pipe endpoints are dynamically bound to peer endpoints at runtime. Peer endpoints correspond to available peer network interfaces (e.g., a TCP port and associated IP address) that can be used to send and receive message. JXTA pipes can have endpoints that are connected to different peers at different times, or may not be connected at all.

Pipes are virtual communication channels and may connect peers that do not have a direct physical link. In this case, one or more intermediary peer endpoints are used to relay messages between the two pipe endpoints.

Pipes offer two modes of communication, point-to-point and propagate, as seen in Figure 11. The JXTA core also provides secure unicast pipes, a secure variant of the point-to-point pipe.

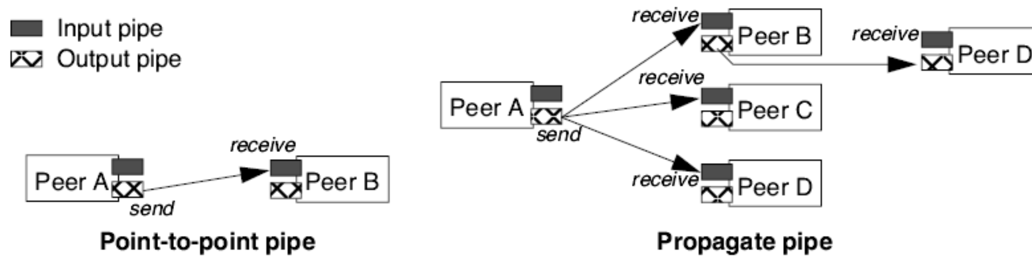


Figure 11 Point-to-point and propagate pipes [28].

Messages

A message is an object that is sent between JXTA peers; it is the basic unit of data exchange between peers. Messages are sent and received by the Pipe Service and by the Endpoint Service. Typically, applications use the Pipe Service to create, send, and receive messages. (In general, applications are not expected to need to use the Endpoint Service directly. If, however, an application needs to understand or control the topology of the JXTA network, the Endpoint Service can be used.)
A message is an ordered sequence of named and typed contents called message elements. Thus a message is essentially a set of name/value pairs. The content can be an arbitrary type.
The JXTA protocols are specified as a set of messages exchanged between peers. Each software platform binding describes how a message is converted to and from a native data structure such as a Java technology object or a C structure.
There are two representations for messages: XML and binary. The JXTA J2SE platform binding uses a binary format envelop to encapsulate the message payload. Services can use the most appropriate format for that transport (e.g., a service which requires a compact representation for a messages can use the binary representation, while other services can use XML). Binary data may be encoded using a Base64 encoding scheme in the body of an XML message.

The use of XML messages to define protocols allows many different kinds of peers to participate in a protocol. Because the data is tagged, each peer is free to implement the protocol in a manner best-suited to its abilities and role. If a peer only needs some subset of the message, the XML data tags enable that peer to identify the parts of the message that are of interest. For example, a peer that is highly constrained and has insufficient capacity to process some or most of a message can use data tags to extract the parts that it can process, and can ignore the remainder.

Advertisements

All JXTA network resources (such as peers, peer groups, pipes, and services) are represented by an *advertisement*. Advertisements are language-neutral metadata structures represented as XML documents. The JXTA protocols use advertisements to describe and publish the existence of a peer resources. Peers discover resources by searching for their corresponding advertisements, and may cache any discovered advertisements locally.
Each advertisement is published with a lifetime that specifies the availability of its associated resource. Lifetimes enable the deletion of obsolete resources without requiring any centralized control. An advertisement can be republished (before the original advertisement expires) to extend the lifetime of a resource.
The JXTA protocols define the following advertisement types:

- *Peer Advertisement* describes the peer resource. The primary use of this advertisement is to hold specific information about the peer, such as its name, peer ID, available endpoints, and any run-time attributes which individual group services want to publish (such as being a rendezvous peer for the group).

- *Peer Group Advertisement* describes peer group-specific resources, such as name, peer group ID, description, specification, and service parameters.
- *Pipe Advertisement* describes a pipe communication channel, and is used by the pipe service to create the associated input and output pipe endpoints. Each pipe advertisement contains an optional symbolic ID, a pipe type (point-to-point, propagate, secure, etc.) and a unique pipe ID.
- *Module Class Advertisement* describes a module class. Its primary purpose is to formally document the existence of a module class. It includes a name, description, and a unique ID (ModuleClassID).
- *Module Spec Advertisement* defines a module specification. Its main purpose is to provide references to the documentation needed in order to create conforming implementations of that specification. A secondary use is, optionally, to make running instances usable remotely, by publishing information such as a pipe advertisement. It includes name, description, unique ID (ModuleSpecID), pipe advertisement, and parameter field containing arbitrary parameters to be interpreted by each implementation.
- *Module Impl Advertisement* defines an implementation of a given module specification. It includes name, associated ModuleSpecID, as well as code, package, and parameter fields which enable a peer to retrieve data necessary to execute the implementation.
- *Content Advertisement* describes content that can be shared in a peer group. Content can be a file, a byte array, code, or process state.
- *Peer Info Advertisement* describes the peer info resource. The primary use of this advertisement is to hold specific information about the current state of a peer, such as uptime, inbound and outbound message count, time last message received, and time last message sent.
- *Rendezvous Advertisement* describes a peer that acts as a rendezvous peer for a given peer group.

Each advertisement is represented by an XML document. Advertisements are composed of a series of hierarchically arranged elements. Each element can contain its data or additional elements. An element can also have attributes. Attributes are name-value string pairs. An attribute is used to store meta-data, which helps to describe the data within the element.

IDs

Peers, peer groups, pipes and other JXTA resources need to be uniquely identifiable. A JXTA ID uniquely identifies an entity and serves as a canonical way of referring to that entity. Currently, there are six types of JXTA entities which have JXTA ID types defined: peers, peer group, pipes, contents, module classes, and module specifications.
URNs are used to express JXTA IDs. URNs1 are a form of URI that "... are intended to serve as persistent, location-independent, resource identifiers". Like other forms of URI, JXTA IDs are presented as text.
An example JXTA peer ID is:

urn:jxta:uuid-59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903

An example JXTA pipe ID is:

urn:jxta:uuid-59616261646162614E504720503250338E3E786229EA460DADC1A176B69B731504

Unique IDs are generated randomly by the JXTA J2SE platform binding. There are two special reserved JXTA IDs: the NULL ID and the Net Peer Group ID.

9.5 Network Architecture

The JXTA network is an ad hoc, multi-hop, and adaptive network composed of connected peers. Connections in the network may be transient, and message routing between peers is nondeterministic. Peers may join or leave the network at any time, and routes may change frequently.

Peers may take any form as long as they can communicate using JXTA protocols. The organization of the network is not mandated by the JXTA framework, but in practice four kinds of peers are typically used:

- *Minimal peer* A minimal peer can send and receive messages, but does not cache advertisements or route messages for other peers. Peers on devices with limited resources (e.g., a PDA or cell phone) would likely be minimal peers.
- *Simple peer* A simple peer can send and receive messages, and will typically cache advertisements. A simple peer replies to discovery requests with information found in its cached advertisements, but does not forward any discovery requests. Most peers are likely to be simple peers.
- *Rendezvous peer* A rendezvous peer is like any other peer, and maintains a cache of advertisements. However, rendezvous peers also forward discovery requests to help other peers discover resources. Any simple peer can configure itself as a rendezvous peer. Or, a peer can be configured to use one or more rendezvous peers. Each rendezvous peer maintains a list of other known rendezvous peers and also the peers that are using it as a rendezvous. Peers send search and discovery requests to rendezvous peers, which in turn forward requests they cannot answer to other known rendezvous peers. They also forward discovery requests to other peers which are using them as a rendezvous. The discovery process continues until one peer has the answer or the request dies. Messages have a default time-to-live (TTL) of seven hops. Loopbacks are prevented by maintaining the list of peers along the message path. An example configuration is shown in Figure 12. Peer A, B and C are simple peers; peers R1, R2, and R3 are rendezvous peers. Peers A and B are configured to use R1 as their rendezvous, while Peer C is configured with R2 as its rendezvous. When Peer A initiates a discovery or search request, it is initially sent to its rendezvous peer — R1, in this example. If R1 doesn't contain the requested information in its local cache, it forwards the request to its rendezvous peers (R2 and R3 in this example) and also to any additional peers that are using it as a rendezvous (peer B). If any of these peers has the requested information, it returns a response. Simple peers (non-rendezvous peers) do not forward any search or discovery requests they receive. However, if a rendezvous peer does not contain the requested information, it will forward the request to its rendezvous peers and to any peers using it as a rendezvous.

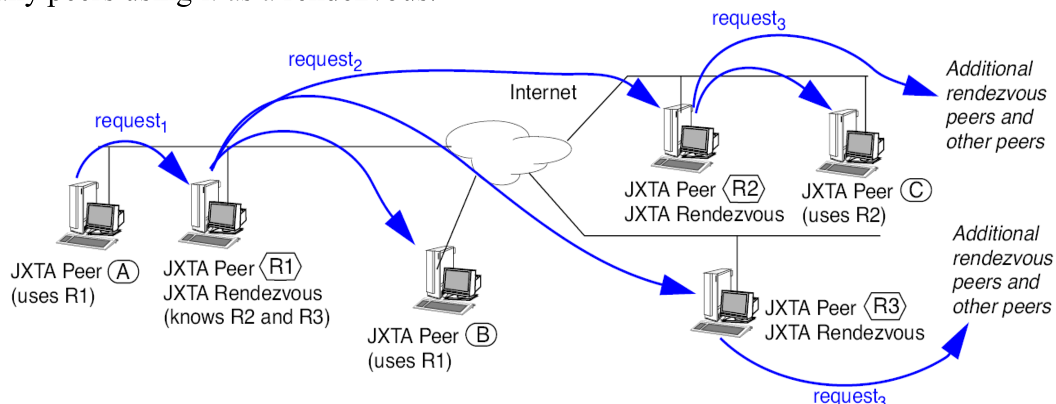


Figure 12 Request propagation via rendezvous peers [28].

- *Relay peer1* A relay peer maintains information about the routes to other peers and routes messages to peers. A peer first looks in its local cache for route information. If it isn't found,

the peer sends queries to relay peers asking for route information. Relay peers also forward messages on the behalf of peers that cannot directly address another peer (e.g., NAT environments), bridging different physical and/or logical networks

Any peer can implement the services required to be a relay or rendezvous peer. The relay and rendezvous services can be implemented as a pair on the same peer.

Firewalls and NAT
A peer behind a firewall can send a message directly to a peer outside a firewall. But a peer outside the firewall cannot establish a connection directly with a peer behind the firewall.
In order for JXTA peers to communicate with each other across a firewall, the following conditions must exist:

- At least one peer in the peer group inside the firewall must be aware of at least one peer outside of the firewall.
- The peer inside and the peer outside the firewall must be aware of each other and must support HTTP.
- The firewall has to allow HTTP data transfers.

These HTTP transfers across the firewall need not be restricted to port 80, although that is the port used by default by most Web browsers and by the current Project JXTA J2SE binding. Figure 13 depicts a typical message routing scenario through a firewall. In this scenario, JXTA Peers A and B want to pass a message, but the firewall prevents them from communicating directly. JXTA Peer A first makes a connection to Peer C using a protocol such as HTTP that can penetrate the firewall. Peer C then makes a connection to Peer B, using a protocol such as TCP/IP. A virtual connection is now made between Peers A and B.

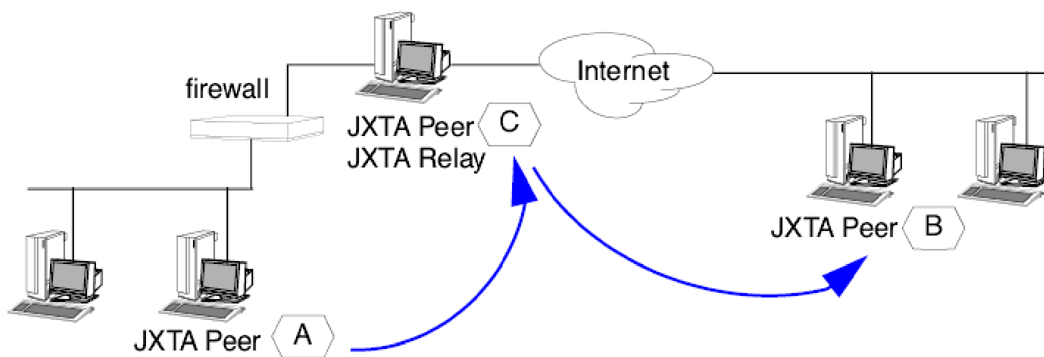


Figure 13 Message routing scenario across a firewall [28].

9.6 JXTA Protocols

JXTA defines a series of XML message formats, or *protocols*, for communication between peers. Peers use these protocols to discover each other, advertise and discover network resources, and communication and route messages.
There are six JXTA protocols:

- *Peer Discovery Protocol (PDP)* used by peers to advertise their own resources (e.g., peers, peer groups, pipes, or services) and discover resources from other peers.
- *Peer Information Protocol (PIP)* used by peers to obtain status information (uptime, state, recent traffic, etc.) from other peers.
- *Peer Resolver Protocol (PRP)* enables peers to send a generic query to one or more peers and receive a response (or multiple responses) to the query. Unlike PDP and PIP, which are used to query specific pre-defined information, this protocol allows peer services to define and exchange any arbitrary information they need.

- *Pipe Binding Protocol (PBP)* used by peers to establish a virtual communication channel, or *pipe*, between one or more peers.
- *Endpoint Routing Protocol (ERP)* used by peers to find routes (paths) to destination ports on other peers. Route information includes an ordered sequence of relay peer IDs that can be used to send a message to the destination (for example, the message can be delivered by sending it to Peer A which relays it to Peer B which relays it to the final destination).
- *Rendezvous Protocol (RVP)* used by peers to propagate messages within a peer group.

All JXTA protocols are asynchronous, and are based on a query/response model. A JXTA peer uses one of the protocols to send a query to one or more peers in its peer group. It may receive zero, one, or more responses to its query.
For example, a peer may use PDP to send a discovery query asking for all known peers in the default Net Peer Group. In this case, multiple peers will likely reply with discovery responses. In another example, a peer may send a discovery request asking for a specific pipe named “aardvark”. If this pipe isn’t found, then zero discovery responses will be sent in reply.
JXTA peers are not required to implement all six protocols; they only need implement the protocols they will use. The current Project JXTA J2SE platform binding supports all six JXTA protocols.
The Java programming language API is used to access operations supported by these protocols, such as discovering peers or joining a peer group.

9.7 Universal Resource Binding and Rendezvous

The Project JXTA network provides an universal resource binding mechanism called the *resolver* to perform all resolution operations found in traditional distributed systems, such as resolving a peer name into an IP address (DNS), binding an IP socket to a port, locating a service via a Directory service (LDAP), or searching for content in a distributed file system (NFS). In Project JXTA, all resolution operations are unified under the simple discovery of one or more advertisements. All binding operations are implemented as the discovery or search of one or more XML documents. Project JXTA does not specify how the search of advertisements is performed. Each peergroup can tailor its resolver implementation to use a decentralized, centralized, or hybrid search approach to match the peergroup’s requirements. The resolver provides a generic infrastructure for services to send and propagate resolver queries and to receive responses. The resolver performs authentication and verification of credentials and drops invalid messages.
The Project JXTA network provides a bootstrapping resolver service based on *Rendezvous* peers. Rendezvous peers are well-known peers that have agreed to cache a large number of advertisements in support of a peergroup. Rendezvous conceptually corresponds to town square where people gather for exchanging and trading information. A peergroup can have as many rendezvous peers as needed. Each peergroup has its own set of rendezvous peers and any peer can potentially become a rendezvous peer. Secure peergroups may enforce a trust model before delegating rendezvous capability to a peer. Rendezvous provide the minimum infrastructure for efficiently building and bootstrapping high-level discovery and search services. For instance, via rendezvous, peers can be discovered to support a more advanced search mechanism. High-level search services are expected to provide more efficient search mechanisms, because they may have a better knowledge of the peergroup content topology distribution [29][30][35]. The rendezvous peer infrastructure provides a low-level discovery mechanism to discover advertisements while providing hooks that allow high-level discovery and search services to participate in the advertisement search process. Peers maintain special relationship with their rendezvous peers. Rendezvous peers determine the set of peers that should receive advertisement query requests. Rendezvous peers can propagate requests to the other known rendezvous peers of a peergroup within the limits of loop detection and message lifetime.

9.8 Peer Monitoring and Metering

Peer monitoring means the capability to closely keep track of a (local or remote) peer's status, control the behavior of a peer, and to respond to actions on the part of a peer. This capability is very useful when a peer network wants to offer premium services with a number of desirable properties such as reliability, scalability, and guaranteed response time. For example, a failure in the peer system must be detected as soon as possible so that corrective actions can be taken. It is sometimes better to shut down an erratic peer and transfer its responsibilities to another peer.

Peer metering means the capability to accurately account for a peer's activities, in particular its usage of valuable resources. Such a capability is essential if the network economy is to go beyond flat-rate services. Even for providers offering flat rate services, it is to their advantage to be able to collect data and analyze usage patterns in order to be convinced that a flat rate structure is sustainable and profitable.

JXTA currently approaches monitoring and metering through the Peer Information Protocol, where a peer can query another peer for data such as up time and amount of data handled.

Obviously, security is central to peer monitoring and metering. A peer may choose to authenticate any command it receives. It may also decide to not answer queries from suspect sources.

9.9 Services

A service denotes a set of functions that a provider offers. A peer can offer a service by itself or in cooperation with other peers. A service provider peer publicizes the service by publishing a service advertisement. Other peers can then discover this service and make use of it. Each service has a unique ID and name that consists of a canonical name string and a series of descriptive keywords that uniquely identifies the service.

Sometimes, a service is well-defined and widely available such that a peer can just use it. Other times, special code may be needed in order to actually access a service. For example, the way to interface with the service provider may be encoded in a piece of software. In this case, it is most convenient if a peer can locate an implementation that is suitable for the peer's specific runtime environment. Of course, if multiple implementations of the same service are available, then peers hosted on systems with Java runtime environments can use Java programming language implementations while native peers to use native code implementations. Service implementations can be pre-installed into a peer node or loaded from the network. The process of finding, downloading, and installing a service from the network is similar to performing a search on the Internet for a web page, retrieving the content of the page, and then installing the required plug-in to work with the page. Once a service is installed and activated, pipes may be used to communicate with the service.

A peer service that executes only on a single peer is a peer service while a service that is composed of a collection of cooperating instances of the service running on multiple peers is a peer group service. A peer group service can employ fault tolerance algorithms to provide the service at a higher level of availability than that a peer service can offer.

Although the concept of a service is orthogonal to that of a peer and a peer group, a group formed using the JXTA platform may require a minimum set of services needed to support the operation of the group.

10. Globus

The software toolkit developed within the Globus project [15] comprises a set of components that implement basic services for security, resource location, resource management, data access, communication, and so on.

The Globus toolkit provides a "bag of services", from which developers can select to build custom tailored tools or applications on top of them. These services are distinct and have well defined

interfaces, and therefore can be integrated into applications/tools in an incremental fashion. The Globus architecture is illustrated in Figure 14.

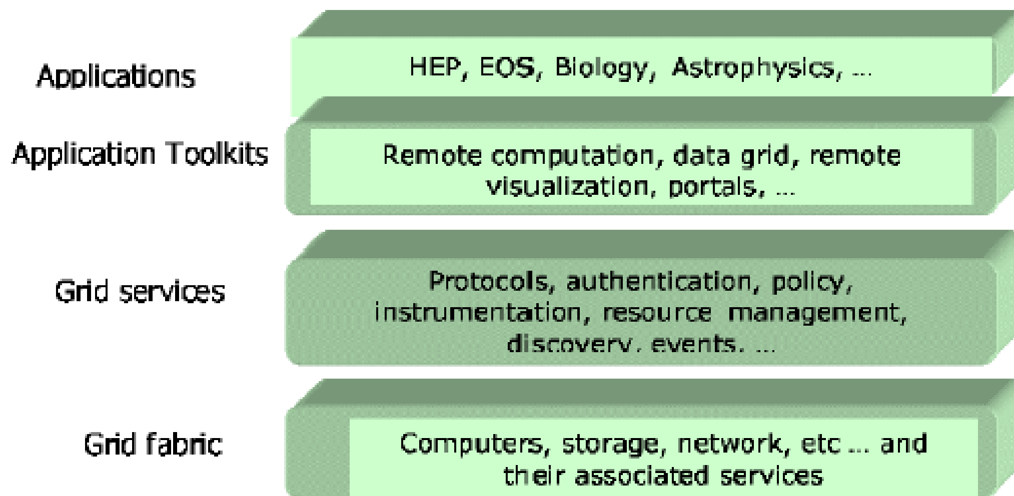


Figure 14 The Globus structure.

The first layer, the Grid Fabric, comprises the underlying systems, computers, operating systems, local resource management systems, storage systems, etc. The components of the Grid Fabric are integrated by Grid Services. These are a set of modules, providing specific services (information services, resource management services, remote data access services, etc.): each of these modules has a well-defined interface, which higher level services use to invoke the relevant mechanisms, and provides an implementation, which uses low-level operations to implement these functionalities. Examples of the services provided by the Globus toolkit are:

- The Grid Security Infrastructure, GSI
- The Grid Information Service, GIS
- The Globus Resource Allocation Manager GRAM

Application toolkits use Grid services to provide higher-level capabilities, often targeted to specific classes of applications. The fourth layer comprises the custom, tailored applications built on services provided by the other three layers. The Globus toolkit uses standards whenever possible, for both interfaces and implementations.

10.1 GIS: the Globus information service

The Information Service (IS) plays a fundamental role in the DataGrid environment, since resource discovery and decision making is based upon the information service infrastructure. Basically an information service is needed to collect and organize, in a coherent manner, information about grid resources and status and make it available to consumer entities.

The Globus *Grid Information Service* (GIS) provides an infrastructure for storing and managing static and dynamic information about the status and components of computational grids. The real and specific problem that underlies the Grid concept is *coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations (VO).*

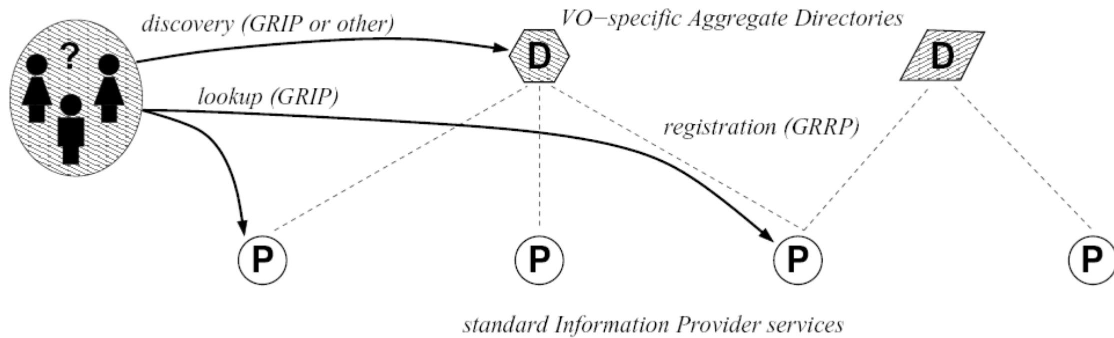


Figure 15 Architecture overview.

The Grid information service architecture (Figure 15) comprises two fundamental entities: highly distributed *information providers* and specialized *aggregate directory* services. Information providers form a common, VO-neutral (infrastructure providing access to detailed, dynamic information about Grid entities. For example, a provider for a compute resource might provide information about the number of nodes, amount of memory, operating system version number and load average; a provider for a running application might provide information about its configuration and current status. Aggregate directories provide often specialized, VO-specific views of federated resources, services, and so on. For example, a directory intended for use by a broker might list the computers available to a VO organized by operating system type; another directory, intended to support application monitoring, might keep track of running applications.

An information provider is defined as a service that speaks two basic protocols. The GRid Information Protocol (GRIP) is used to access information about entities, while the GRid Registration Protocol (GRRP) is used to notify aggregate directory services of the availability of this information. *These two protocols are the fundamental building blocks on which the architecture is based*. For an entity to be known to VO participants, it must either speak these protocols directly (hence being its own information provider) or interact with some other entity that acts as an information provider on its behalf. An aggregate directory is a service that uses GRRP and GRIP to obtain information (from a set of information providers) about a set of entities, and then replies to queries concerning those entities. An aggregate directory can itself adopt GRIP as the protocol by which others query it (and, for that matter, GRRP as the protocol that it uses to notify others of its existence), but it is not obliged to do so: in fact, such directories can support arbitrary data models, query languages, and protocols.

The Grid Information Protocol

A user, or more frequently an aggregate directory or other program, uses GRIP to obtain information from an information provider about the entity(s) on which the provider possesses information. Because an information provider can possess information on more than one entity, GRIP supports both discovery and enquiry. Discovery is supported via a *search* capability. For example, consider an information provider that maintains information on a set of workstations. A broker might then perform a search on that provider to obtain a set of results that roughly match a given criteria. From the set of discovered resources, enquiry can be used to refine the set of resources upon which a broker may schedule. Enquiry corresponds to a direct *lookup* of information: the enquiry supplies the resource name and the provider returns the resource description. Subscription (i.e., a request that results in the subsequent delivery of a sequence of updates) can be an important enquiry mode, and should be supported.

The standard Lightweight Directory Access Protocol (LDAP) as the protocol for GRIP is adopted. LDAP defines a data model, query language, and wire protocol. The *data model* uses *object classes* with named types to characterize resources, and hierarchical *distinguished names* to name resources within the information provider. The *query language* supports search, lookup, and (via recently

proposed extensions) subscription operations. A filter can be used in all cases to specify a set of criteria to be matched. A subset of attributes associated with an entity can be retrieved (reducing the amount of information that must be transmitted. This query language has been used within MDS-1. Finally, the LDAP *protocol* supports a query-reply exchange, with the query specifying a search, lookup, or subscription, and the reply comprising the specified fields of any matching object(s).
The LDAP query language also has its limitations. In particular, it cannot specify relational "joins," i.e., operations that allow data from several different typed entities to be combined to yield a new composite entity. A join operation can be supported when needed via an optimized discovery service, using techniques such as those found in current LDAP-based meta-directory servers.

The Grid Registration Protocol

GRRP complements GRIP by defining a notification mechanism that one service component can use to "push" simple information about its existence to another element of the information services architecture. For example, GRRP is used by an information provider to notify a aggregate directory of its availability. It is a soft-state protocol, meaning in the context that state established at a remote location by a notification (e.g., an index entry for an information provider) may eventually be discarded unless refreshed by a stream of subsequent notifications. Such protocols have the advantages of being both resilient to failure (a single lost message does not cause irretrievable harm) and simple (no reliable "de-notify" protocol message is required).
Each GRRP message contains the name of the service that is being described (i.e., a URL to which GRIP messages can be directed), the type of notification message, and timestamps that determine the interval over which the notification should be considered to hold. The GRRP definition does not specify the underlying transport: it is designed to un over an unreliable transport, but a reliable transport can also be used. GRRP provides a discoverer with an *unreliable failure detector*. A discoverer can decide at a certain point (e.g., after a certain amount of time has passed without a registration message being received from a producer) that the producer has failed or become inaccessible.

Aggregate Directory Services

Having defined GRIP and GRRP there are no technical limitations on what sorts of indices or naming, search, and monitoring strategies directories may maintain, or on what data models, query languages, and protocols they may support. Nevertheless, there are significant benefits to adopting standard data models, query languages, and protocols within aggregate directories, as otherwise users and programs have to write different code to query each directory variant that might be produced. In practice a Grid information system will see a small number of standard aggregate directory structures.

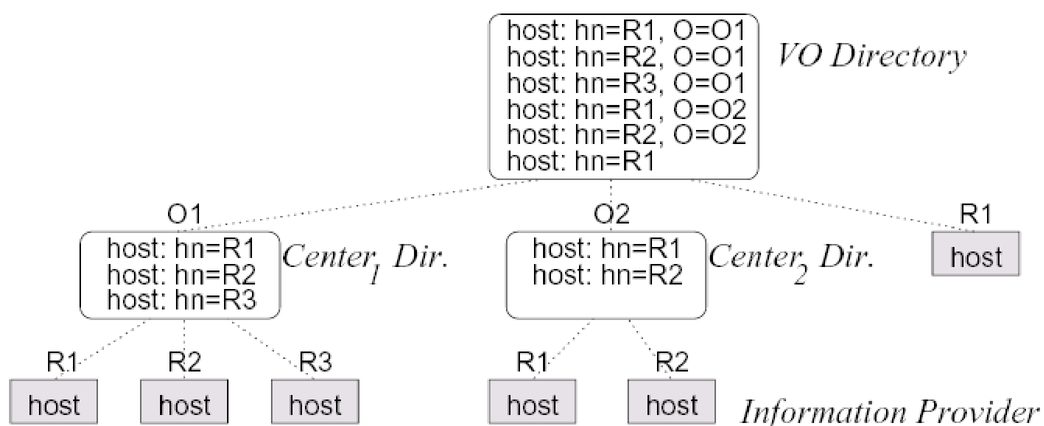


Figure 16 Hierarchical discovery.

Figure 16 shows how a hierarchical discovery service can be structured as a network of aggregate directories. Each directory uses the GRIP data model, query language, and protocol, and acts as an information provider that contains information about all of the resources beneath it in the hierarchy. Directories use GRRP to register with higher-level
directories to construct the hierarchy. Such aggregate directories could also use lossy aggregation techniques, as in the Service Discovery Service, which hashes descriptions and summarizes hashes via Bloom filtering.
This hierarchical discovery structure conveniently mirrors the typical decomposition of VO administration, with multiple site administrators coordinating with the VO service administrator(s). Each site administrator can maintain their local aggregate directory and register it with directories maintained by the VOs in which the site participates.
Local aggregate directories can use GRRP to register with VO directories just as information providers use GRRP to register with local aggregate directories, as described in the preceding section.

Monitoring and Other Applications

GRIP and GRRP can be used to construct a variety of other services and applications, concerned with such things as brokering, monitoring, application adaptation, troubleshooting, and performance diagnosis.
GRIP is designed to support both delivery models and hence to support both discovery and monitoring. In *pull* mode, a query-response exchange supports on-demand access to information; in *push* mode, an initial subscription request requests subsequent asynchronous delivery. However, even given this flexibility, there are information delivery roles for which GRIP is ill-suited. The retrieval of archival information can require the support of more powerful database query interfaces, to reduce search costs over a continuously growing mountain of data. Similarly, various flavors of event delivery system can provide specialized synchronization and reliability properties (e.g., source-ordered delivery to multiple recipients, or exactly-once delivery).
For those exceptional situations where GRIP is not adequate, the architecture allows for extensions in two ways:

- *GRIP extension.* Resources may offer additional information delivery capabilities beyond those provided by GRIP. For example, an information provider that interfaces to a large archive might implement protocol extensions to support richer relational queries.
- *Service publication.* GRRP and GRIP are designed to permit discovery of other Grid resources and services. For example, a high data rate network monitor may deliver information via a specialized, binary-encoded push protocol. The information provider for this monitor can indicate that this protocol is supported, and provide the information needed to subscribe to it.

Configuring Information Services

Manual configuration. Users or system administrators can configure information providers with the addresses of directories, or directories with the names of providers. This approach has obvious scalability problems, but is practical in the case of small and/or long-lived VOs.
Automated discovery based on a hierarchical discovery service. New discovery services can potentially be configured via searches of a hierarchical discovery service, if one exists.
Automated discovery based on other information services. Services such as SLP, DNS, or UDDI can be used to assist with configuration. For example, clients can use SLP to locate a default local directory from which to initiate VO resource discovery. Multicast techniques may sometimes be appropriate.

10.2 GRAM: the Globus resource allocation manager

The Globus resource management architecture, illustrated in Figure 17, is a layered system, in which high level services are built on top of a set of local services.

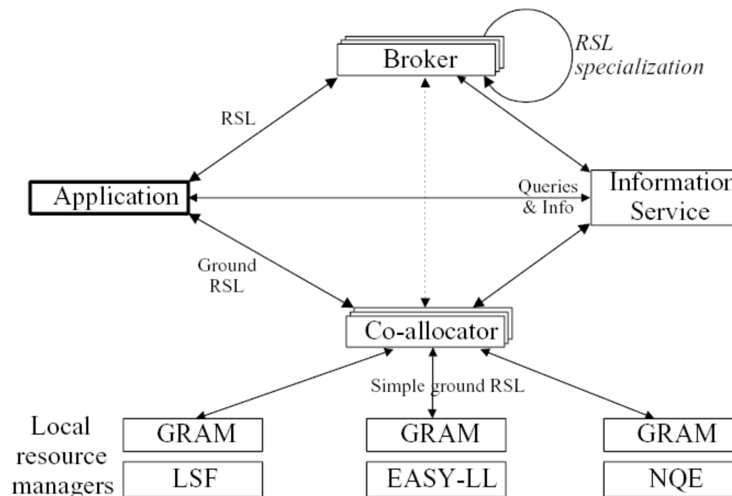


Figure 17 The Globus resource management architecture.

In this architecture, an extensible *resource specification language* (RSL) is used to communicate requests for resources between components: from applications to resource brokers, resource co-allocators and resource managers. At each stage in this process, information about resource requirements is coded as an RSL expression by the application or refined by one or more resource brokers and co-allocators; information about resource availability and characteristics can be is obtained from an information service.

Resource brokers are responsible for taking high-level RSL specifications and transforming them into more concrete specifications through a process called *specialization*. As illustrated in Figure 18, multiple brokers may be involved in servicing a single request, with application-specific brokers translating application requirements into more concrete resource requirements, and different resource brokers being used to locate available resources that meet those requirements.

Transformations effected by resource brokers generate a specification in which the locations of the required resources are completely specified. Such a *ground request* can be passed to a *co-allocator*, which is responsible for coordinating the allocation and management of resources at multiple sites. Resource co-allocators break a multirequest (that is, a request involving resources at multiple sites) into its constituent elements and pass each component to the appropriate *resource manager*. As discussed in Section 5, each resource manager in the system is responsible for taking a RSL request and translating it into operations in the local, site-specific resource management system.

The *information service* (the MDS-2 described in the section above) is responsible for providing efficient and pervasive access to information about the current availability and capability of resources. This information is used to locate resources with particular characteristics, to identify the resource manager associated with a resource, to determine properties of that resource, and for numerous other purposes during the process of translating high-level resource specifications into requests to specific managers.

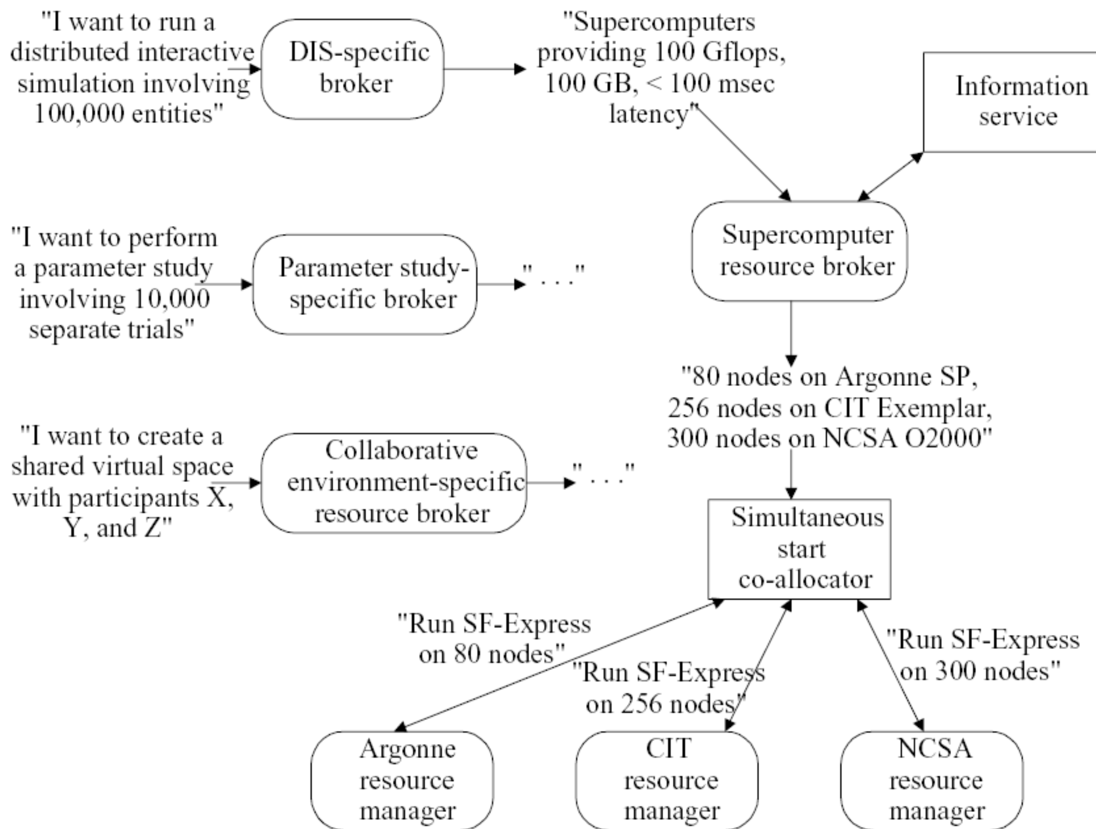


Figure 18 Globus resource management architecture.

Local Resource Management

The lowest level of the resource management architecture is named local resource managers and the implementation of this entity in the architecture is called a Globus Resource Allocation Manager (GRAM); is responsible for

1. processing RSL specifications representing resource requests, by either denying the request or by creating one or more processes (a \job") that satisfy that request;
2. enabling remote monitoring and management of jobs created in response to a resource request; and
3. periodically updating the MDS information service with information about the current availability and capabilities of the resources that it manages.

As indicated above, a GRAM is intended to serve as the interface between a wide area Meta-computing environment and an autonomous entity able to create processes, such as a parallel computer scheduler. Notice that this means that a resource manager need not correspond to a single host or a specific computer, but rather to a service that acts on behalf of one or more computational resources. This use of local scheduler interfaces was first explored in the software environment for the I-WAY networking experiment, but is extended and generalized here significantly to provide a richer and more flexible interface.

A resource specification passed to a GRAM is assumed to be ground: that is, to be sufficiently concrete that the GRAM can identify local resources that meet the specification without further interaction with the entity that generated the request. A particular GRAM implementation may achieve this goal by scheduling resources itself or, more commonly, by mapping the resource specification into a request to some local resource allocation mechanisms. Hence, the GRAM API plays for resource management a similar role to that played by IP for communication: it can co-exist with local mechanisms, just as IP rides on top of Ethernet, FDDI, or ATM networking technology.

The GRAM API provides functions for submitting and for canceling a job request, and for asking when a job (submitted or not) is expected to run. An implementation of the latter function may use queue time estimation techniques. When a job is submitted, a globally unique *job handle* is returned that can then be used to monitor and control the progress of the job. In addition, a job submission call can request that the progress of the requested job be signaled asynchronously to a supplied *callback URL*. Job handles can be passed to other processes, and callbacks do not have to be directed to the process that submitted the job request. These features of the GRAM design facilitate the implementation of diverse higher-level scheduling strategies. For example, a high-level broker or co-allocator can make a request on behalf of an application, while the application monitor the progress of the request.

GRAM Implementation

The GRAM implementations have the structure shown in Figure 19.

The principal components are the GRAM client library, the gatekeeper, the RSL parsing library, the job manager, and the GRAM reporter. The Globus security infrastructure (GSI) is used for authentication and for authorization.

The *GRAM client library* is used by an application or a co-allocator acting on behalf of an application. It interacts with the GRAM gatekeeper at a remote site to perform mutual authentication and transfer a request, which comprises a resource specification, a callback (described below), and a few other components that are not relevant to the current discussion.

The *gatekeeper* is an extremely simple component that responds to a request by doing three things: performing mutual authentication of user and resource, determining a local user name for the remote user, and starting a job manager which executes as that local user and actually handles the request. The first two security-related tasks are performed by calls to the Globus security infrastructure (GSI), which handles issues of site autonomy and substrate heterogeneity in the security domain. In order to start the job manager, the gatekeeper must run as a privileged program: on Unix systems, this is achieved via suid or inetd. However, because the interface to the GSI is small and well defined, it is easy for organizations to easily and approve the gatekeeper code. In fact, the gatekeeper code has successfully undergone security reviews at a number of large supercomputer centers. The mapping of remote user to locally recognized user name minimizes the amount of code that must run as a privileged program; it also allows us to delegate most authorization issues to the local system.

A *job manager* is responsible for creating the actual processes requested by the user. This task typically involves submitting a resource allocation request to the underlying resource management system, although if no such system exists on a particular resource, a simple fork may be performed. Once processes are created, the job manager is also responsible for monitoring the state of the created processes, notifying the callback contact of any state transitions, and implementing control operations such as process termination. A job manager terminates once the job for which it is responsible has terminated.

The *GRAM reporter* is responsible for storing into MDS various information about scheduler structure (e.g., whether the scheduler supports reservation and the number of queues) and state (e.g., total number of nodes, number of nodes currently available, currently active jobs, and expected wait time in a queue). An advantage of implementing the GRAM reporter as a distinct component is that MDS reports can continue even when no gatekeeper or job manager is running: for example, when the gatekeeper is run from inetd.

GRAM implementations have been constructed for six local schedulers to date: Condor, LSF, NQE, Fork, EASY, and LoadLeveler. Much of the GRAM code is independent of the local scheduler and so only a relatively small amount of scheduler-specific code needed to be written in each case. In most cases, this code comprises shell scripts that make use of the local scheduler's user-level API. State transitions are mostly handled by polling, because this proved to be more reliable than monitoring job process by using mechanisms provided by the local schedulers.

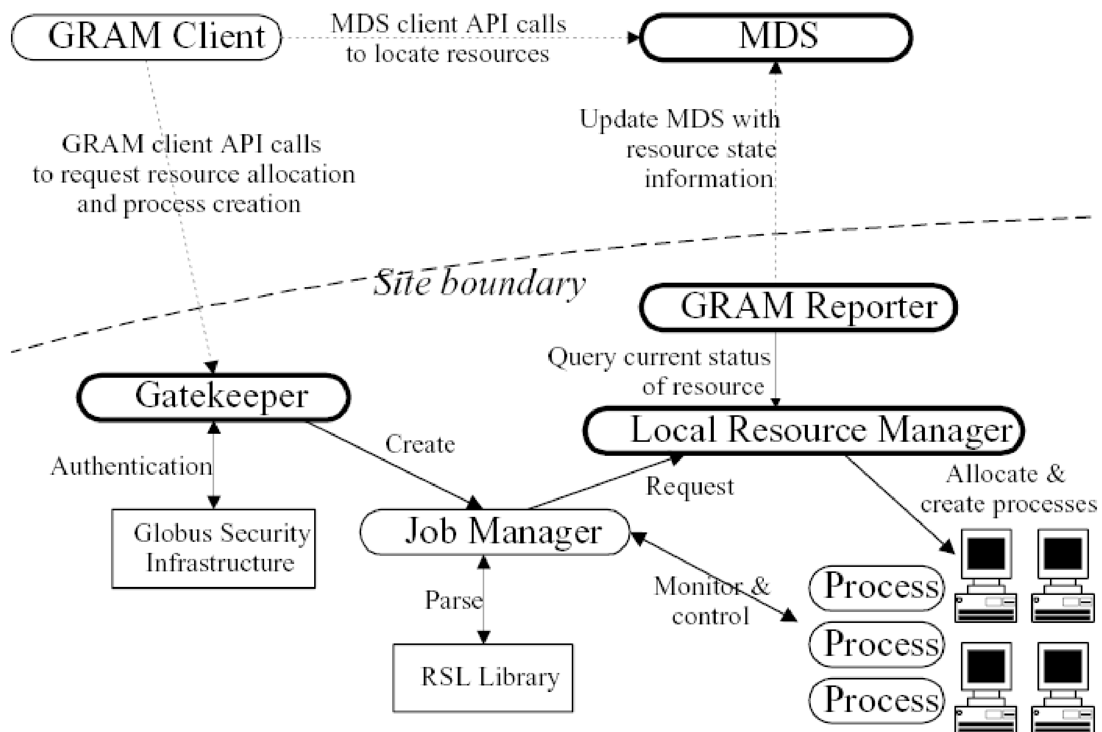


Figure 19 Major components of the GRAM implementation.

10.3 GridFTP

The GridFTP protocol and family of tools were born out of a realization that the Grid environment needed a fast, secure, efficient, and reliable transport mechanism. The Globus Project surveyed available protocols and technologies, implemented some prototypes, and settled on using FTP and its existing extensions as a base, and then extending it again to add missing required functionality. The result is a protocol and a family of tools that provide the following features:

- **Grid Security Infrastructure (GSI) and Kerberos support:** Robust and flexible authentication, integrity, and confidentiality features are critical when transferring or accessing files. GridFTP supports both GSI and Kerberos authentication, with user controlled setting of various levels of data integrity and/or confidentiality.
- **Third-party control of data transfer:** In order to manage large data sets for large distributed communities, it is necessary to provide third-party control of transfers between storage servers. GridFTP provides this capability by adding GSSAPI security to the existing third-party transfer capability defined in the FTP standard.
- **Parallel data transfer:** On wide-area links, using multiple TCP streams can improve aggregate bandwidth over using a single TCP stream. This is required both between a single client and a single server, and between two servers. GridFTP supports parallel data transfer through FTP command extensions and data channel extensions.
- **Striped data transfer:** Partitioning data across multiple servers can further improve aggregate bandwidth. GridFTP supports striped data transfers through extensions defined in the Grid Forum draft.
- **Partial file transfer:** Many applications require the transfer of partial files. However, standard FTP requires the application to transfer the entire file, or the remainder of a file

starting at a particular offset. GridFTP introduces new FTP commands to support transfers of regions of a file.

- **Support for reliable data transfer:** Reliable transfer is important for many applications that manage data. Fault recovery methods for handling transient network failures, server outages, etc., are needed. The FTP standard includes basic features for restarting failed transfer that are not widely implemented. The GridFTP protocol exploits these features, and substantially extends them.
- **Manual control of TCP buffer size**: This is a critical parameter for achieving maximum bandwidth with TCP/IP. The protocol also has support for automatic buffer size tuning, but it has not yet implemented anything in the code. There were communications with both NCSA and LANL to see if it makes sense to integrate work they are doing in this area into the code.
- **Integrated Instrumentation**: The protocol calls for restart and performance markers to be sent back. It is not specified how often, and this is something that has to be address shortly.

11. OGSA

The *Open Grid Services Architecture* (OGSA) [16] integrates key Grid technologies (including the Globus Toolkit) with Web services mechanisms to create a distributed system framework based around the Grid service. A *Grid service instance* is a (potentially transient) service that conforms to a set of conventions (expressed as WSDL interfaces, extensions, and behaviors) for such purposes as lifetime management, discovery of characteristics, notification, and so forth. Grid services provide for the controlled management of the distributed and often long-lived state that is commonly required in sophisticated distributed applications. OGSA also introduces standard factory and registry interfaces for creating and discovering Grid services.
A deeper description of the OGSA can be found in [31 32 33].

11.1 The OGSA Service Model

A basic premise of OGSA is that everything is represented by a *service*: a network enabled entity that provides some capability through the exchange of messages. Computational resources, storage resources, networks, programs, databases, and so forth are all services. This adoption of a uniform service-oriented model means that all components of the environment are virtual.
More specifically, OGSA represents everything as a *Grid service*: a Web service that conforms to a set of conventions and supports standard interfaces for such purposes as lifetime management. This core set of consistent interfaces, from which all Grid services are implemented, facilitates the construction of hierarchal, higher-order services that can be treated in a uniform way across layers of abstraction.
Grid services are characterized (*typed*) by the capabilities that they offer. A Grid service implements one or more *interfaces*, where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages. Grid service interfaces correspond to portTypes in WSDL. The set of portTypes supported by a Grid service, along with some additional information relating to versioning, are specified in the Grid service's *serviceType*, a WSDL extensibility element defined by OGSA.

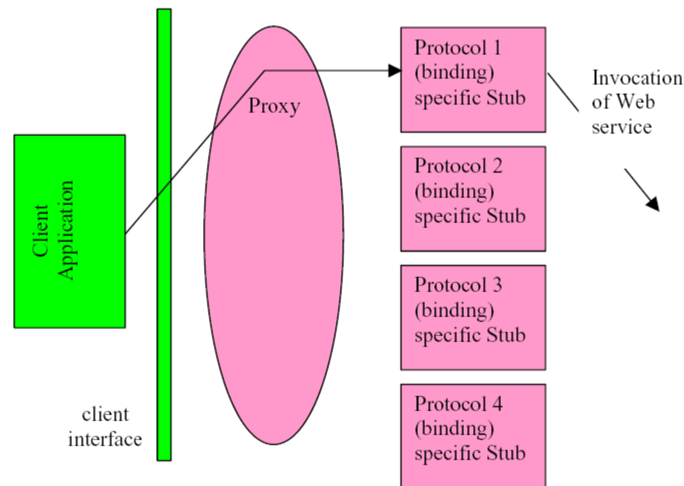


Figure 20 A possible client-side runtime architecture.[1]

Figure 20 depicts a possible client-side architecture for OGSA. In this approach, there is a clear separation between the client application and the client-side representation of the Web service (proxy), including components for marshalling the invocation of a Web service over a chosen binding. In particular, the client application is insulated from the details of the Web service invocation by a higher-level abstraction: the client-side interface. Various runtime tools can take the WSDL description of the Web service and generate interface definitions in a wide range of programming language specific constructs (e.g. Java interfaces). This interface is a front-end to specific parameter marshalling and message routing that can incorporate various binding options provided by the WSDL. Further, this approach allows certain efficiencies, for example, detecting that the client and the Web service exist on the same network host, and therefore avoiding the overhead of preparing for and executing the invocation using network protocols.

OGSA services can be created and destroyed dynamically. Services may be destroyed explicitly, or may be destroyed or become inaccessible as a result of some system failure such as operating system crash or network partition. Interfaces are defined for managing service lifetime.

Because Grid services are dynamic and stateful, a way to distinguish one dynamically created service instance from another is needed. Thus, every Grid service instance is assigned a globally unique name, the *Grid service handle (GSH),* that distinguishes a specific Grid service instance from all other Grid service instances that have existed, exist now, or will exist in the future. Grid services may be upgraded during their lifetime, for example to support new protocol versions or to add alternative protocols. Thus, the GSH carries no protocol or instance-specific information such as network address and supported protocol bindings. Instead, this information is encapsulated, along with all other instance-specific information required to interact with a specific service instance, into a single abstraction called a *Grid service reference* (GSR). Unlike a GSH, which is invariant, the GSR(s) for a Grid service instance can change over that service's lifetime. Each GSR has an explicit expiration time, and OGSA defines mapping mechanisms for obtaining an updated GSR.

[1] This figure has been selected from [33].

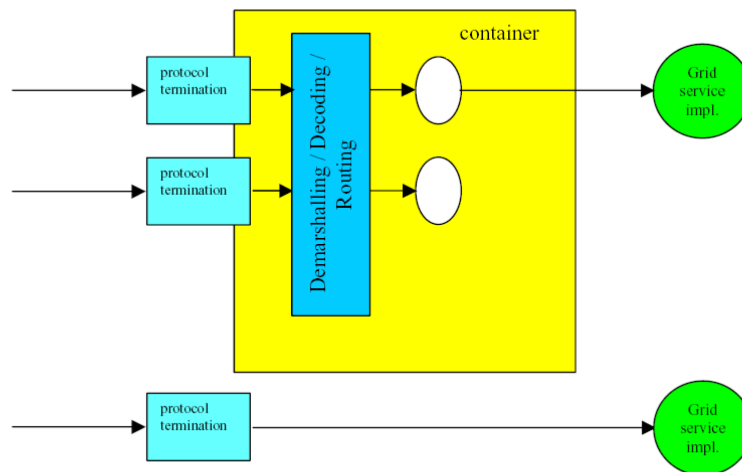


Figure 21 Two alternative approaches to the implementation of a service.[2]

Figure 21 illustrates two different approaches to the implementation of argument de-marshalling functions. The assumption is that, as is the case for many Grid services, the invocation message is received at a network protocol termination point (e.g., an HTTP servlet engine), which converts the data in the invocation message into a format consumable by the hosting environment. The top of Figure 21 illustrates two Grid services (the ovals) associated with container-managed components (for example EJBs within a J2EE container). Here, the message is dispatched to these components, with the container frequently providing facilities for de-marshalling and decoding the incoming message from a format (such as an XML/SOAP message) into an invocation of the component in native programming language. In some circumstances (the lower oval), the entire behaviour of a Grid service is completely encapsulated within the component. In other cases (the upper oval), a component will collaborate with other server-side executables, perhaps through an adapter layer, to complete the implementation of the Grid services behaviour. At the bottom of Figure 21, another scenario wherein the entire behaviour of the Grid service, including the de-marshalling/decoding of the network message, has been encapsulated within a single executable is depicted. Although this approach may have some efficiency advantages, it provides little opportunity for reuse of functionality between Grid service implementations.

OGSA defines a class of Grid services that implement an interface that creates new Grid service instances: this interface is called *Factory* interface and a service that implements this interface is a *factory*. The *Factory* interface's CreateService operation creates a requested Grid service and returns the GSH and initial GSR for the new service instance.

The *Factory* interface does not specify how the service instance is created. One common scenario is for the factory interface to be implemented in some form of hosting environment (such as .NET or J2EE) that provides standard mechanisms for creating (and subsequently managing) new service instances.

The introduction of transient service instances raises the issue of determining the service's lifetime: that is, determining when a service can or should be terminated so that associated resources can be recovered. In normal operating conditions, a transient service instance is created to perform a specific task and either terminates on completion of this task or via an explicit request from the requestor or from another service designated by the requestor. In distributed systems, however, components may fail and messages may be lost. One result is that a service may never see an expected explicit termination request, thus causing it to consume resources indefinitely.

OGSA addresses this problem through a soft state approach in which Grid service instances are created with a specified lifetime. The initial lifetime can be extended by a specified time period by explicit request of the client or another Grid service acting on the client's behalf (subject of course

[2] This figure has been selected from [33].

to the policy of the service). If that time period expires without having received a re-affirmation of interest from a client, either the hosting environment or the service instance itself is at liberty to terminate the service instance and release any associated resources.
Many OGSA mechanisms derive from the Globus Toolkit: in particular, the factory (GRAM gatekeeper), registry (GRAM reporter and MDS-2), use of soft-state registration (MDS-2), secure remote invocation with delegation (GSI), and reliable remote invocation (GRAM). The primary differences relate to how these different mechanisms are integrated.
A comparison between Globus Toolkit and OGSA architectures in shown in Figure 22.

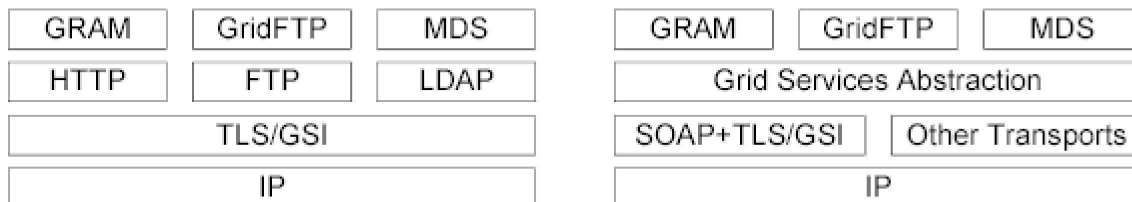


Figure 22 Architecture of Globus Toolkit (left) and OGSA (right) [32].

12. Comparison

12.1 Resource discovery

- Gnutella doesn't have a discovery system, but uses a broadcast message passing. When a user search a resource, the user node broadcast a query to all nodes in its neighbor set N. Each node in N forwards the query to its neighbors. There is a Time to Live field to handle the number of hops a query can cross. NOTE: the resource are files!
- Freenet uses a hashing system to associate a key to each file. When a user make a query, each node checks its own store, and if it finds the file, returns it. Otherwise, the node forwards the request to the node in its table with the closest key to the one requested. NOTE: the resources are files.
- JXTA provides an universal resource binding mechanism, the resolver, to perform all resolution operations found in traditional distributed systems such as resolving a peer name into an IP address, locating a service via a Discovery Service. All binding operations are implemented as the discovery or search of one or more XML documents. A bootstrapping resolver service is based on Rendezvous peers: well-known peers that cache a large number of advertisements. NOTE: the resources are services.
- Globus uses the Globus Information Service to support both discovery and enquiry. The standard Lightweight Directory Access Protocol is adopted. LDAP defines a query language that permits search and lookup.

12.2 Resource management

- Freenet uses a mechanism that prioritizes space allocation by the frequency of request per file. Each node orders the files in its data store by the time of last request and when a new file arrives (and there isn't space left) the node deletes the least recently used files. This improves response time and prevents overloading when the file is not request.
- In JXTA there is a peer monitoring system: this means the capability to closely keep track of a (local or remote) peer's status, control the behavior of a peer, and to respond to actions on the part of a peer. JXTA offers also a peer metering system: the capability to account for a

peer's activity, in particular its usage of resources. JXTA approaches monitoring and metering through the Peer Information Protocol.

- Globus uses the Globus Resource Allocation Manager (GRAM): a layered system, in which high level services are built on top of a set of local services. The resource broker is responsible for taking high –level resource specification and transforming them into more concrete specification. Such requests can be passed to co-allocator, which is responsible for coordinating the allocation and management of resources at multiple sites. Resource co-allocators pass each component of a request to the appropriate resource manager (Figure 19). The responsible for providing efficient and pervasive access to information about the current availability of resources is the Information Service.

12.3 Resources Description

- In Freenet each resource has a content-hash key (CHK): a low-level data-storage key that is generated by hashing the contents of the file to be stored. This key gives every file an unique absolute identifier. There is also the signed-subspace key (SSK) that sets up a personal namespace that anyone can read but only its owner can write to.
- All JXTA network resources (peers, peer groups, services, ...) are represented by an advertisement: language-neutral metadata structures represented as XML documents. Each advertisement is published with a lifetime that specifies the availability of its associated resource. Lifetimes enable the deletion of obsolete resources without requiring any centralized control. All resources have a JXTA ID that uniquely identify the entity and serves as a canonical way of referring to that entity.
- In the Globus OGSA a Grid service instance is a service that conforms to a set of conventions (expressed as WDSL interfaces, extensions and behaviors) for such purposes as lifetime management, discovery of characteristic, notification, and so forth. Each Grid service as a globally unique name, the Grid service handle, that distinguishes a specific instance from all other instances.

13. Security analysis

13.1 Freenet and Gnutella

As said before, Freenet was designed with anonymity in mind and consequently the security was not a priority. Gnutella is in a quite similar position in the sense that Gnutella is just a specification of a protocol designed for anonymous file exchange with no security considerations. Nevertheless, it is conceivable to implement such a protocol with adding a security system like the Globus Security Infrastructure.

13.2 Globus

In this document only the Globus Security Infrastructure (GSI) will be discussed. In addition, only the second version will be considered. This is due to the lack of documentation dealing with the third version of GSI which is currently in developing phase.
The asymmetric cryptography (which is a public key cryptography) is the basis of the functionality of GSI. Three points are important about GSI concepts :

- The security in communications between elements of the network.
- The support of security across organizational boundaries. This implies the existence of a centralized security manager system.
- The support of "single sign-on" for users of the network and delegation of credentials for computations that involve multiple resources.

The GSI certificates are needed for the identification of each members participating in the network. Such a certificate is defined with four attributes : a subject name (the identity of the user or the object), a public key, the identity of the Certificate Authority (which sign the certificate to ensure that the identity and the public key of the user are trustable) and the digital signature of this CA. The authentication of identity is made by a trusted third-party (the Certificate Authority). The GSI certificates are encoded in the x509 certificate format (see at the end of this document).
Certificates make both parties of a communication trustable. By trusting the CA that signed the other's certificate, each member can prove to the other that they are who they say they are. This is the *mutual authentication*. The Secure Sockets Layer (SSL, which is also known as the new IETF standard name : Transport Layer Security or TLS – see at the end of this document) is used by GSI as the protocol for mutual authentication.
Each user of the GSI has a private key at his disposal. This private key is stored in a file in the user's computer, and, for security considerations, this file is encrypted. The only way to use this private key in the GSI is to provide the correct pass phrase in order to decrypt the file.
At last, GSI provides a delegation capability. That consists of an extension of the standard SSL protocol which reduces the number of times the user must identify him with entering pass phrase. The need to re-enter the user's pass phrase can be avoided by using a *proxy*. The proxy is a new certificate (containing a public key and the owner's identity slightly modified) and a private key. This new certificate is not signed by a CA but by the owner himself. Proxies have limited lifetimes after what they can't be trustable.

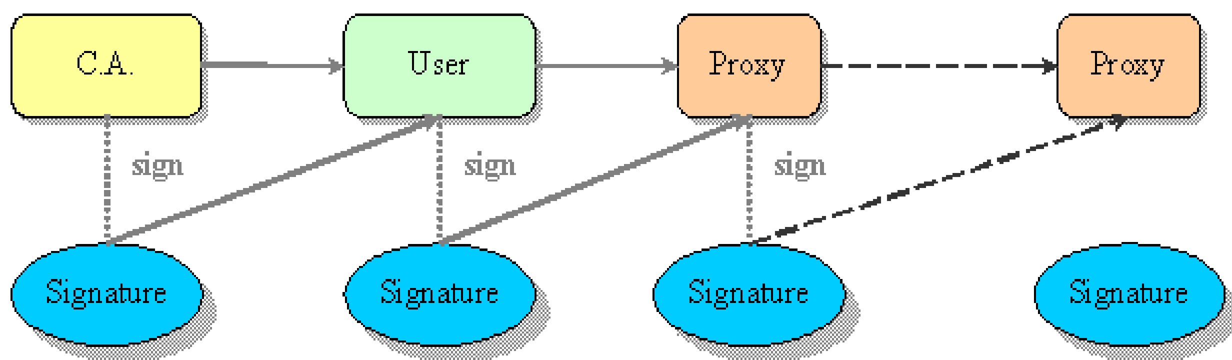


Figure 23 The use of proxy in the delegation.

When proxies are used, the mutual authentication process differs slightly (Figure 23). The remote party receives the proxy's certificate (which is signed by the owner of the proxy) and the owner's certificate. The owner's public key is used in order to validate the signature of the proxy certificate. The CA's public key is then used to ensure the validity of the signature of the owner's certificate.

Using GSI without Globus

GSI is a set of tools and libraries provided with the Globus Toolkit. The GSI allows the use of x509 certification for secure authentication within a network. Two applications authorize the use of authentication portion of Globus Toolkit : GSI-enabled Secure Shell (SSH) and GSI-enabled FTP (gsiftp).

13.3 JXTA

JXTA adopts a security model that relies on existing, trusted technologies. This security system is a consequence of three choices made by the project JXTA team :

- the adoption of the Transport Layer Security (or TLS, an emerging protocol for the secure transport of information),
- the exploitation of end-to-end transport independence of JXTA protocols,
- and the use of X509.V3 digital certificates (see at the end of this document for more details about these security technologies).

Poblano

Trust is one of the primordial principles in relationship between peers within a network. On a peer-to-peer network (or more generally on a decentralized network), users can see the origin of information. They can also communicate their "opinion" on both the information they have received and the peers that are its source. All these opinions can be collected and evaluated in order to be used as guidelines for searching information, recommending information sources, thus creating decentralized and personalized "webs of trust". Such a decentralized trust model implemented on a peer-to-peer network can be compared to the notion of trust between human beings in "real" life.
The goal of Poblano [36] is to establish such a decentralized trust model on the JXTA platform. A second application of Poblano is to build a recommendation system for security purposes. In current trust or reputation models (like in FreeHaven [37] or Publius [38]), the degree of trust of a peer is calculated with parameters such as honesty, reliability, performance, etc. Poblano introduces the notion of user's interest or keyword in the sense that the evaluation of the trust for the peer is based on interests and keywords (see figure 2).

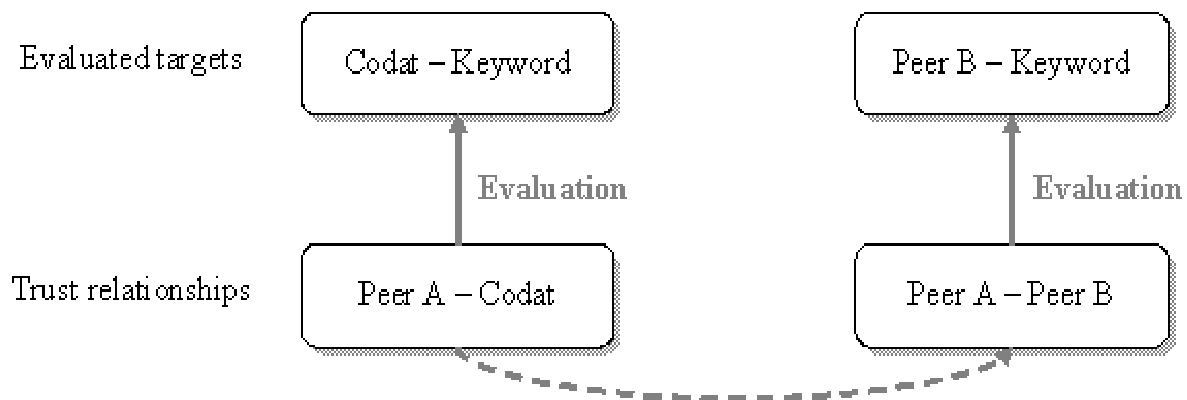


Figure 24 Trust relationship in Poblano.

The first thing for user A is to evaluate his trust on a *codat* (a codat is a general term that covers static as well as dynamic or executable data which is locally or remotely stored). The user builds then a trust relationship with that codat. The results of "interests" evaluation on the set of codats received by A from B will be used to evaluate the A's trust of B as a source for the given collection of keywords.

13.4 Security technologies

TLS

TLS stands for Transport Layer Security. It supports reliable private connections between peers. The TLS protocol (of which specification is available at http://www.ietf.org/rfc/rfc2246.txt) is

composed of two layers : the TLS Record Protocol and the TLS Handshake Protocol. The first of these protocols provides connection security in two ways :

- The connection is private, and symmetric cryptography is used for data encryption (using DES, RC4, etc.)
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC, these computations use secure hash functions, like SHA, MD5…

The other protocol, the TLS Handshake Protocol, provides connection security that has three properties :

- The peer's identity can be authenticated using asymmetric, or public key, cryptography (e.g., RSA, DSS, etc.). This authentication can be optional.
- The negotiation of a shared secret is secure.
- The negotiation is reliable.

Last point, the TLS is application protocol independent, so higher level protocols can layer on top of TLS transparently.

X509.V3 digital certificates

Two concepts are primordial in electronic security : the concepts of authentication and non-repudiation. Both are intended to ensure the identity of parties to a transaction.
Many security system use the principle of public and private keys (for example, the RSA security system uses public/private key identification). The goal of a certificate is to certify such a public key. In order to use a public key, it is preferable to be sure of the owner of this key. One method is to create a direct confidence relation with its owner, like in PGP system. Another method is for all members of a transaction to entrust the creation of a certificate to a third-party called Certificate Authority. X509 is a standard which defines certificates format issued by such a Certificate Authority.

13.5 Summary

	JXTA	**Globus** (1)	**Freenet / Gnutella**
Level of security	High security	High security	No security in the protocol specification.
Security basis		Asymmetric cryptography (2)	
Certificates	X509v3 certificates	GSI Certificate (x509)	
Certificates Authorities (CA)	Not fixed. CA can be centralized or peers can become their own CA.	The use of proxy in the delegation capability authorizes to ignore CA.	
Authentication	TLS	SSL / TLS (3)	
Communications	End-to-end transport independence of JXTA protocols.	No encryption by default.	
Other available security systems	All security services and applications provided by Sun and the JXTA community. *Poblano* model.	Delegation and "single sign-on" capabilities (4)	
Notes	Security services or applications can be added on JXTA core.	The GSI is only a set of libraries and tools for security concerns.	Freenet and Gnutella are just protocol specifications.
Available	Sun has provided many	Few documents about	Few available

documentation	documents about JXTA technologies.	Globus itself are available.	documents. Essentially protocol specification.

(1)	This concerns only the Globus Security Infrastructure (GSI) version 2 because of the lack of documentation dealing with the third version.
(2)	Asymmetric cryptography is also known as public key cryptography.
(3)	The Transport Layer Security (TLS) is the new standard name of Secure Sockets Layer (SSL).
(4)	The delegation capability is an extension of standard SSL protocol which reduces the number of time the user must identify him with entering pass phrase.

14. Workflow management systems

The main goal of a data grid is to distribute the information on a collection of widely heterogeneous, largely distributed, and loosely coupled computing environments. The deployment of large and heterogeneous distributed environments involves several issues concerning how interrelated tasks can be efficiently executed and closely monitored. In this context, workflow tools can be of great interest [39].

Workflow management systems (WFMSs) have their origin in office automation and they typically aim to coordinate business processes. The same approach can be used to coordinate tasks and resources over a distributed environment such as a middleware. The workflow management allows to describe: units of work (for instance tasks running on different machines), the data to be processed, and the control flow (i.e. how the different units have to interact each other with respect to a set of constraints and dependences).

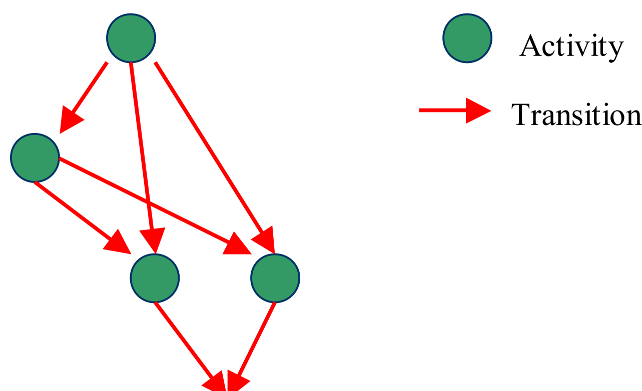
A workflow management tool must be able to provide the user the necessary support for cooperative work with other users that could use different software and database systems.

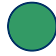
The central issue for a workflow management system is to define:

who must do what, when and how

The definition of the workflow is delivered by specifying a set of activities and transitions. An activity (i.e. the *what* part) represents something to be done: sending an e-mail, updating a database, and so on. On the other hand, a transition represents the *when* part, that is the appropriate sequence of activities for a process. The *how* part is implemented by associating to every activity a specified application able to carried out the job. The *who* part is determined by the user assigned to carry out the activity, through its application. The entity assigned to an activity could be a person as well as an automatic system.

The answer to the question: "*who must do what, when and how*" is delivered by a formal process definition. First of all activities and transitions have to be defined; a bubbles and arrows formalism can be used to model the *what* and *when* parts.





Transitions connect activities together. Connecting two activities A with B, a transition states that as soon as A is finished B has to be started.

An activity can be of three different kinds: application, sub-process or routing. The work described by an activity can be carried out by an application or a better definition of the job to be performed can be obtained by a sub-process. Sometimes an activity has to be handle the work flow routing in the process, and in this case the activity simply describes what has to be done next.

Two kind of guards are generally associated to an activity: incoming and outgoing guards. The incoming guards can be set to *and* (means that all the activities that lead to this activity have to be completed in order to enable this activity to work) or *xor* (means that just one of the activities that lead to this activity has to be completed in order to enable this activity to work) for two different behaviors. On the other hand, outgoing guards (collecting outgoing transitions getting out of an activity) can be set to a*nd* (means that this activity will trigger the work of all the activities it connects to; doing this the flow in the process will be split in parallel) or *xor* (means that this activity will trigger the work of just one of the activities it connects to, depending on condition evaluation)

Moreover, the work of an activity could be triggered in dependence on the start mode. If the start mode is set to *automatic* the activity will run its application as soon as an instance workitem reaches it; if the start mode is set to *manual* the user intervention is wait to start the activity application.

Although this generally describes the WFMS behavior, a taxonomy of workflow management systems is usually based on their "nature": administrative, collaborative, ad-hoc, and so on [40].
In general, *administrative* workflows refer to bureaucratic processes where the steps to follow are well-established and are based on known rules. *Ad hoc* workflows address exceptions and unique situations. The situation might not be exceptional but each particular instance is unique. The third class, *collaborative* workflows, is mainly characterized by the number of participants involved and their interactions. Unlike other types of workflow that assume continual forward progress, a collaborative workflow might involve several iterations over the same step until the participants reach some agreement; it might even involve going back to an earlier stage. *Production* workflows are the high end of these systems. They can be characterized as the implementation of mission-critical business processes—that is, those directly related to the organization's function. Credit and loan applications and insurance claims are typical examples.

14.1 Workflow languages

In WFMSs the environment is described in term of work units (they could be pre-existent software modules), autonomous services in a network, and data repositories (for instance databases). The work flow defines how the single units interact and the linguistic specification of these interaction is said: *workflow language* (or coordination languages). Workflow languages have to be able to provide an useful and feasible support to implement WFMSs.
A set of attributes of workflow languages can be identified:

- Advanced control mechanisms. The workflow language must provide a set of statements to specify execution state dependencies and data dependencies; moreover, it has to be possible to check the execution state of every task and to control the work flow according with the state values.

- Support for system integration. A work flow language must provide statements to support the communication and the communication control among the activities. Fault-tolerance mechanism should be provided.
- Object model. The work flow specification can be a hard task and it should be possible to decompose it into sub-problems (top-down approach). OOP (Object Oriented Programming) techniques can be very useful in this context. Objects encapsulates data (variables) and behavior (methods); objects can be seen as units that offer a well-defined interface via their methods
- Transaction capability. The terms task and activity could be replaced by the term transaction. Typically a transaction must guarantee a set of properties: atomicity, consistency, isolation, and durability of the task execution (ACID property [41]). However, the ACID properties can be too strict for heterogeneous environments; for this reason, several advanced transaction models have been provided. For example, the Flex transaction model [42] was designed especially to fulfill requirements of heterogeneous systems and for multidatabase systems by relaxing ACID properties

Two kind of workflow languages can be identified: general purpose, special purpose. General purpose workflow languages, generally are extensions of general purpose languages (for instance the C language) by new coordination toolkits in order to provide them of workflow coordination capabilities. A survey of general purpose languages can be found in [43]. In particular, the C&Co language is presented; it is an extension of the C language where the concept of communication variable is introduced. A communication variable is just like other variables in C but it can be shared among parallel and distributed processes. Another example of language extended to support workflow capabilities is Triana [44] in the context of the GridOneD project [45]. In particular Triana has been extended in order to support services to be used within a Grid context. Maybe the first general purpose software to support the communication among processes in a WFMS is Linda [46 47]. Linda provides a simple and effective way to separate computation and communication tasks. The basic idea of Lind is to implement a shared space of *tuples* where processes can insert and from which can read/delete data. Operation on a *tuple* are blocking and hence the synchronization is automatically delivered. Several improvements of Linda have been proposed in order to overcome issues concerning transaction and fault-tolerance capability.

Although general purpose languages allow to integrate extensions in a smooth way, a completely new language (targeted for workflow management) has generally the advantage of making possible a better software design. An example is Manifold [48]; Manifold is a coordination language for orchestration of the communications among independent, cooperating processes in a massively parallel or distributed application. The fundamental principle underlying Manifold is the complete separation of *computation* from *communication.*

It is worthwhile to stress all workflow/coordination languages are generally oriented to tackle issues concerning massively parallel computation (problems potentially addressed by computational grids) or transactions over largely distributed database systems (problems potentially addressed by data grids)

15. Distributed data management systems

This section aims to provides an overview of techniques and technologies could be useful to address distributed data management issues.

The need for managing large amount of distributed data stems from a spread field of disciplines that can generate Terabytes or Petabytes of persistent data [49]. Applications producing this amount of

data are generally named “data intensives” as applications requiring high computational power are defined computationally intensive.
Examples of applications are: data analysis in High Energy Physics (HEP), climate modeling, earth and solar observation, and so on. Projects such as the European DataGrid Project [50 51], PPDG [52] and GriPhyN[53] explicitly deal with distributed data management. The Spitfire Project (within the European DataGrid Project) aims to provide a uniform way to access many RDBMSs (Relational Database Management Systems) through grid web services [54].
The EGSO project belongs to this kind of applications and ideas, experiences, and software developed above all within the European DataGrid Project could be of great interest during the system design phase.

The problem of distributed data management can coarsely analyzed from two different points of view: distributed databases and data grids (with the term grid is meant a middleware able to provide an unified interface to distributed data).

The first issue to be addressed in a distributed data management system is the data replication. In this context, the term distributed identifies a set of machines interconnected by a WAN (Wide Area Network). A centralized approach could strongly affect the whole system performance, therefore a data replication over the WAN is often necessary.

Distributed databases mainly address the two following issues: replica synchronization and synchronous and asynchronous replication. Very seldom cost function and large amount of data (Petabytes scale) problems are considered. Moreover, an unique data access method is generally available independent of the amount of the data accessed.
On the other hand, from the data grid point of view there are several efforts to enable fast and efficient file transfers, a replica catalogue for managing files and some more replica management functionalities based on files (at the grid level, the file is typically considered as the lowest granularity of replication as the replica manager does not need to know the structure of the file).
Research on the data grid aims to provide an intermediate (replication) layer able to manage file replications. This layer has to take into account that local systems over the grid generally can use different database management systems. For this reason, the replication layer should be responsible of replication and synchronization among different DBMSs (global transactions) whereas each DBMS is responsible for transactions on local data (local transactions).
A possible approach to data replication used by Globus involves the use of the LDAP directory service (The Lightweight Directory Access Protocol LDAP [55] is a protocol for accessing online directory services; it runs directly over TCP, and can be used to access a standalone LDAP directory service or to access a directory service that is back-ended by X.500) according with a DBMS for managing local files. Assuming that the local files catalogue is provided by the local DBMS (not supporting files replication) a global replica catalogue has to be introduced in order to take into account multiple copies of a file (a *name space* has to be also added). The combination of LDAP and a DBMS provides another advantage concerning the possibility to store heterogeneous data. However, this mechanism needs a database backend for LDAP where the file information is stored and concurrency mechanisms have to be established. Moreover, in order to guarantee the synchronization, each site storing the replica catalogue information has to run locally a LDAP server.

The problem of consistency of replicas is strictly related to data replication. A replica is not a simple copy of the original “document” as a replica has a logically connection with the original. Two kinds of replication can be classified:

- Synchronous
- Asynchronous

A fully synchronous replica fulfills the highest degree of consistency as a local transition has immediately propagated to all the other replicas (or at least at the majority of them). This degree of consistency can strongly affect the whole system performance and it should be applied only when strictly necessary. A middleware needs to "external communication mechanisms" and it is rather difficult that synchronous consistency can be provided; on the other hand, a DBMS could deliver specific locking mechanisms extended to replicas providing better performance.
In order to overcome the lack of performance of synchronous replications (above all when write operations are involved) research focuses on asynchronous mechanisms:

1. *Primary-copy approach*. Just one primary copy exists and replicas are considered as secondary copies [56]. If a write operation refers a secondary copy the request is passed to the primary copy that, after the update, propagates the changes to all secondary copies.
2. *Epidemic approach*. Operations can be related to every single replica but a separate process has to compare version information and propagates the updates [57].
3. *Subscription approach*. A site can contain data without caring of consistency. When a copy is updated, only a certain number of sites is informed following the classical mechanism of subscription. If a site has not subscribed is itself responsible to get updated information. This approach provides the "minimum level of consistency" but, on the other hand, delivers a great flexibility as sites can decide the data to be imported and the data to be filtered. An implementation of this strategy can be found in the Grid Data Management Pilot (GDMP) [58].

Consistency mechanisms generate some messages as a global transaction involves all sites. A global transaction not necessarily need to create locks at each site, but al least notification messages have to be sent. Two kinds of message can be identified: control message and data transfer. The former is used to synchronize the sites, whereas the latter denotes a real transfer of data (or metadata) from one site to another one.

16. References

1. **Computational grids** *Fox, F.; Gannon, D.*
   Computing in Science & Engineering , Volume: 3 Issue: 4 , July-Aug. 2001
2. **A taxonomy and survey of grid resource management systems for distributed computing**
   Krauter K.; Buyya R.;Maheswaran M..
   Softw. Pract. Esper., 2002, 32
3. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmag01/html/netpeers.asp/
4. http://choices.cs.uiuc.edu/2k/
5. http://apples.ucsd.edu/
6. http://bond.cs.ucf.edu/
7. http://www.cs.wisc.edu/condor/
8. http://www.eu-datagrid.org/
9. http://www.globus.org/
10. http://javelin.cs.ucsb.edu/
11. http://legion.virginia.edu/
12. http://icl.cs.utk.edu/netsolve/
13. http://ninf.apgrid.org/
14. http://punch.ecn.purdue.edu/
15. http://www.globus.org/
16. The *Open Grid Services Architecture* (OGSA), http://www.globus.org/ogsa/
17. http://www.gnutella.com/
18. *Gnutella Protocol* documented by Clip 2 DSS,
    http://www.gnutellahosts.com/GnutellaProtocol04.pdf
19. *FASD: A Fault-tolerant, Adaptive, Scalable, Distributed Search Engine*, Amr Z. Kronfol, Princeton University, 2002.
20. *Scalability issues in large peer-to-peer networks—a case study of gnutella***.** Jovanovic, M. A., Annexstein, F. S., et al. Tech. rep., University of Cincinnati, 2001.
21. *Why gnutella can't scale. no really*. J. Ritter, http://www.darkridge.com/ jpr5/doc/gnutella.html.
22. Kazaa. http://www.kazaa.com.
23. http://freenetproject.org/
24. *Protecting Free Expression Online with Freenet*, Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wiley, *IEEE Internet Computing* 6(1), 40-49 (2002).
25. *Freenet: A Distributed Anonymous Information Storage and Retrieval System.* Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, in *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, LNCS 2009, ed. by Hannes Federrath. Springer: New York (2001).
26. Project JXTA, *www.jxta.org*
27. *Project JXTA: A Technology Overview.* L. Gong http://www.jxta.org/project/www/
28. *Project JXTA: Java[tm] Programmers Guide.* http://www.jxta.org/project/www/
29. *Project JXTA Virtual Network.* B. Traversat, M. Abdelaziz, http://www.jxta.org/project/www/

30. *A Scalable Content Addressable Network*. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, ACM SIGCOM, 2001
31. *Grid Services for Distributed System Integration*. I. Foster, C. Kesselman, J. Nick, S. Tuecke Computer, 35(6), 2002.
32. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.* I. Foster, C. Kesselman, J. Nick, S. Tuecke; http://www.globus.org/ogsa/ June 22, 2002.
33. *Grid Service Specification*. C. Kesselman; S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, Draft 2, 7/17/2002. http://www.globus.org/ogsa/
34. *Building Peerto-Peer Systems with Chord, a Distributed Lookup Service.* F. Dabek, E. Brunskill, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan, 2001
35. *The Semantic Web*. T. Berners-Lee, J. Hendler, and O. Lassila, http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html
36. *Poblano. A distributed trust model of peer-to-peer networks.* R. Chen and W. Yaeger, Sun Microsystems, Inc.
37. *Trust economies in the Free Haven Project*. B.T. Sniffen.
38. *Publius : A robust, tamper-evident, censorship-resistant web publishing system*. M. Waldman, A.D. Rubin and L.F. Cranor.
39. *WFMS: The Next Generation of Distributed Processing Tools,"Advanced Transaction Models and Architectures,* G.Alonso and C.Mohan, S. Jajodia and L. Kerschberg, eds., Kluwer Academic, Hingham, Mass.,1997, pp. 35-62.
40. *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure,* D. Georgakopoulos, M. Hornick and A. Sheth, *Distributed and Parallel Databases,* Vol. 3,No. 2,Apr.1995,pp. 119-153
41. *Concurrency Control and Recovery in Database Systems.* Ph. Bernstein, V. Hadzilacos, and N. Goodman, Addison-Wesley, 1987.
42. *A multidatabase transaction model for InterBase.* A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz, In Proceedings of The 16th International Conference on Very Large Data Bases, 1990.
43. *General Purpose Work Flow Languages.* A. Forst, E. Kühn, and O. Bukhres, International Journal on Parallel and Distributed Databases, Vol. 3, No. 2, 1995.
44. *Triana:* http://triana.co.uk/
45. *GridOneD:* http://www.gridoned.org/

46. *Linda, the Portable Parallel.* R. Bjornson, N. Carriero, D. Gelernter, and L. Jerrold, Technical Report 520, Yale University Department of Computer Science, Jan. 1988.
47. *How to Write Parallel Programs: A Guide to the Perplexed.* N. Carriero and D. Gelernter, Technical Report 628, Yale University Department of Computer Science, May 1988.
48. *Manifold:* http://www.cwi.nl/projects/manifold/
49. *Distributed Database Management Systems and the Data Grid.* H. Stockinger, 18th IEEE Symposium on Mass Storage Systems, 2001.
50. *The European DataGrid Project:* http://www.cern.ch/grid/
51. *Data Management in International Data Grid Project.* W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, K. Stockinger, 1st IEEE/ACM International Workshop on Grid Computing, December 2000.
52. *Particle Physics Data Grid (PPDG):* http://www.ppdg.net
53. *GriPhyN:* http//www.griphyn.org
54. *Project Spitfire – Towads Grid Web Services Databases*. W.H. Bell, D. Bosio, W. Hoscheck, P. Kuunszt, G. McCance, M. Silander, Informational Document, Global Grid Forum 5, Edimburgh, July 2002.
55. *The SLAPD and SLURPD Administrators Guide University of Michigan Release 3.3*: http://www.umich.edu/~dirsvcs/ldap/doc/guides/slapd/
56. *Replication and Consistency: Being Lazy Helps Sometimes*. Y. Breitbart, H. Korth, In Proceedings of 16th ACM Sigact/Sigmond Symposium on the Principles of Database Systems, 1997.
57. *Epidemic Algorithms in Replicated Databases*. D. Agrawal, A.E. Abbadi, R. Steinke, PODS 1997.
58. *Grid Data Management Pilot (GDMP)*: *A Tool for Wide Area Replication*. A. Samar, H. Stockinger, IASTED International Conference on Applied Informatics, 2001.