CHAPTER 13

Jini and JXTA Based Lightweight Grids

Kurt Vanmechelen, Jan Broeckhove,
Gunther Stuer, Tom Dhaene
Department of Mathematics and Computer Sciences, University of Antwerp,
BE-2020 Antwerp, Belgium

CONTENTS

1. Introduction . . . 1
2. Characteristics of LWG . . . 2
3. A Comparison of Jini and JXTA . . . 4
   3.1. Vision . . . 4
   3.2. Scope . . . 5
   3.3. Resource Discovery and Publication . . . 5
   3.4. Service Interaction . . . 8
   3.5. Group Concept . . . 10
   3.6. Security . . . 10
   3.7. Flexibility . . . 11
   3.8. Ease-of-Use . . . 11
4. Contributions of Jini and JXTA to an LWG Middleware . . . 11
   4.1. Limited Administrative Resources . . . 11
   4.2. Heterogeneity . . . 12
   4.3. Volatility . . . 12
   4.4. Scale . . . 13
   4.5. Openness . . . 13
   4.6. Hostility . . . 13
5. Overview of Jini and JXTA in Lightweight Grid Systems . . . 14
   5.1. LWGs and Jini . . . 14
   5.2. LWGs and JXTA . . . 15
6. Conclusion . . . 15
   References . . . 16

1. INTRODUCTION

The benefits of distributed computational systems are well established, and numerous software architectures and toolkits have evolved in recent years to support this mode of computing. Most of these systems however are rather limited in scope. Some, such as

Engineering the GRID
Edited by B. Di Martino, J. Dongarra, A. Hoisie, L. T. Yang, and H. Zima
Pages (1–17)
ISBN: 1-58883-038-1
Copyright © 2006 by American Scientific Publishers
All rights of reproduction in any form reserved.

SETI@Home [1] are targeted at specific research projects, while others, such as cluster middlewares, are usually confined to single administrative domains.

More generalized metacomputing systems, or Grids [2], have gained tremendous popularity in recent times because they enable secure, coordinated, resource sharing across multiple administrative domains, networks, and institutions. Their goal is to create virtual organizations (VOs) across institutional and political borders in order to stimulate international cooperation and joint research efforts. These Grids typically consist of a relatively small and static group of, permanently and completely dedicated, high-end nodes. These nodes are typically the supercomputers and clusters which can be found in the participating research centers. Also, the Grid resources are typically not open in the sense that anyone can submit jobs to them. Usually, they are set up to solve only one particular problem, defined in the mission statement of the virtual organization. This model, known as heavyweight Grids (HWG), has been realized in several software toolkits, such as Globus, the Enabling Grids for E-science project in Europe (EGEE), the Large Hadron Collider Computing Grid (LCG), Astrogrid, EU-Datagrid, the Biomedical Informatics Research Network (BIRN), DOE Science Grid, TeraGrid, and NorduGrid. References [3, 4] provide an excellent overview.

However, many applications of smaller magnitude do not require explicit coordination and centralized services for authentication, registration, and resource brokering as is the case in traditional Grid systems. For these applications, a lightweight model in which individuals and organizations share their superfluous resources on a peer-to-peer basis, is more suitable. In this contribution, we will consider the requirements for a lightweight Grid middleware. A number of such lightweight Grids have been realized using the Jini [5] or JXTA [6] frameworks. We will review these frameworks and look into how they can be used to (partially) fulfill some of these requirements.

This chapter is subdivided into five sections and organized as follows. Section 2 will explore the characteristics of the environment in which a lightweight Grid has to operate and outline the resulting requirements for the middleware. The next section will give an overview of Jini and JXTA technologies, comparing their vision and their technological features. In Section 4 we discuss how to relate those features to the middleware requirements formulated in Section 2. The last section presents an overview of some of the projects that have used Jini and/or JXTA for developing (lightweight) Grid systems.

2. CHARACTERISTICS OF LWG

In contrast to their heavyweight counterparts, lightweight Grids broaden the scope of resource sharing by including lower-end computing resources as found in desktop machines or laptops at home or in the office. This results in a new and significantly different environment in which the lightweight Grid has to operate. In this section we discuss the properties of this environment and the consequences for the requirements of the Grid middleware.

Limited administrative resources Heavyweight Grids mainly integrate supercomputers and high-end clusters that are hosted in an institutional context. These systems are maintained and administered by dedicated professional personnel. This lowers the “ease of administration” requirement on heavyweight Grids. Often heavyweight Grids have an important installation and maintenance cost, even for relatively simple applications demands [7]. However, because of the small number of system administrators involved, it is feasible to provide training for the administrators and reduce the burden on end-users. In the lightweight Grid setting, large numbers of people will want to share their resources. However, these individuals will not have the technical expertise nor the time to administer their resources on a day to day basis. Therefore, the middleware should be highly self configurable, self maintainable and very easy to install and administer.

In addition to the larger number of users, a significantly higher number of applications will be deployed on a global lightweight Grid, each with its own requirements concerning the execution environment. Therefore, special attention should be given to the ease of deployment of new applications in the Grid, and automated preparation of the execution environment hosted by the computational resources in support for these applications.

Heterogeneity Another consequence of including general desktop resources is the higher degree of both hardware and software heterogeneity in the Grid fabric. At the software level, the middleware needs to deal with differences in operating systems, installed libraries, local security policies, etc. At the hardware level great variances related to CPU, storage, memory and bandwidth availability have to be dealt with. In heavyweight Grids, homogeneity of the software backplane is sometimes dictated by the strict adherence to OS and middleware package listings that are managed centrally. This is no longer possible in lightweight Grids as resources are expected to be non-dedicated.

Volatility In clusters or heavyweight Grids, the resources are permanently and completely dedicated to the Grid. This is no longer the case for home and office resources. The resource owner has to be able to enforce full control on the conditions under which resources are shared. These can include the requirement of non-obtrusiveness (i.e., only superfluous resources are shared), time based requirements (not during office hours) or virtual organization (VO) specific requirements (support for a limited and dynamic set of applications and virtual organizations). The highly dynamic nature of this mode of sharing results in a high level of volatility in the availability of shared resources. This volatility has a prominent impact on the middleware components that make up the lightweight Grid:

- Because resources leave and join the Grid fabric at higher rates, efficient and dynamic lookup, discovery and monitoring of resources becomes more important.
- More complex scheduling policies and algorithms are required that take into account the resource volatility.
- The need for fault tolerant execution of composite workflows (*failure transparency*) and migration of jobs or services between resources (*migration transparency*) become more important. The same goes for having a persistent service state (*persistence transparency*) and for support of atomic transactions (*transaction transparency*).

Scale Due to the large number of participants in a lightweight Grid, both as resource providers and resource consumers, scalability is of the utmost importance. With large numbers of concurrent nodes, all forms of centralized services have to be approached with caution. Services such as lookup, discovery, resource brokering, scheduling, accounting and monitoring have to be implemented in a highly scalable and distributed fashion. On the other hand, lightweight Grids also have a prominent role in setting up relatively small, institutional Grids. Such Grids pose different requirements on the middleware's services. Consequently, a general purpose lightweight Grid middleware should be highly *customizable* and *modular*, giving the deployer the possibility to selectively assemble different middleware components according to the deployment context.

Openness Heavyweight Grids are typically closed systems in that only a limited number of parties are authorized to submit jobs. The right to consume resources in a heavyweight Grid setting is linked to membership of a particular virtual organization. Enrollment in a VO however, is still largely dependent on human intervention for authenticating the identity of the prospective member. Also, the knowledge resource providers have of the applications that a VO hosts is limited, as is the knowledge resource consumers have of the characteristics and quality of the resources that are generally available. All of this makes joining a Grid and formulating service level agreements for resource sharing, a less than transparent proposition.

The even higher number of users and applications targeted by a lightweight Grid will require a more dynamic and automated form of resource sharing. Service level agreements and collaborations will need to be established on the fly based on the *system's knowledge* of resource provider and consumer characteristics and policies. This requires the necessary accounting and monitoring infrastructure to be in place.

More importantly, in this peer-to-peer lightweight Grid setting, some incentive or reward has to be given to resource providers. This could be achieved by a token based approach. Providers earn tokens for sharing their resources and are able to trade them for remote resources later on. For resource providers that are not consumers, a conversion from tokens to another (hard) currency or service is necessary. A first step in

this direction has been taken by the Compute Power Market project and the Nimrod/G resource broker [8–10].

Hostility A lightweight Grid should be able to operate in an environment where a multitude of independent clients consume the resources delivered by a multitude of independent providers across the internet. This is in effect a hostile environment where resource consumers, owners and their process of information exchange should be protected from malicious actors. Protection of the provider system against malicious consumers or third-parties becomes an important issue. If LWG middleware needs to be easy to install, a strict, but easy to understand, security model should be adopted. Proper authentication and authorization mechanisms should be in place in order to be able to trace user actions and limit their potentially negative consequences.

At present, lightweight Grid systems do not address all of the issues outlined above, often focusing on one particular aspect among the environmental characteristics. According to the taxonomy given in Ref. [11] an LWG operating in the above environment would be characterized as an Autonomic and Lightweight Peer-to-Peer Desktop Grid. The following sections will discuss how Jini and JXTA can contribute to such a Grid.

3. A COMPARISON OF JINI AND JXTA

Clearly the design and implementation of a lightweight Grid system is an extensive undertaking. Leveraging existing technologies or technology components is indispensable. A number of groups have considered Jini [12] or JXTA [13] in the realization of their Grid middleware. Both technologies, developed by Sun Microsystems, have a mission statement that involves enabling or facilitating distributed computing in a dynamic environment.

Jini offers the developer the possibility to partition a user community and its resources into groups, also called federations, and to publish and discover services within the context of a federation. A key focus of the framework is to support the flexible addition and removal of services within such a federation. Jini further exploits the mobile code feature of the Java language and extends the RMI programming model to include dynamic lookup and discovery, support for distributed events, transactions, resource leasing and a new security model. The design of the Jini framework and its resulting programming model are strongly influenced by the recognition of the *Eight Fallacies of Distributed Computing* [14].

The JXTA framework consists of the specification of six XML protocols that enable the creation of dynamic user communities in a heterogenous peer-to-peer environment (P2P). The framework aims to support key peer-to-peer abstractions such as multi-peer communication, peers and peer groups. A *pipe* abstraction is used to represent end-to-end communications between peers over a *virtual overlay network* [15]. This virtual network enables seamless communication between peers in the presence of networking barriers such as firewalls and Network Address Translation, through the use of *relay* peers.

Clearly the thrust of each technology is different and one needs to dig a little deeper to figure out what each of them could possibly contribute to a Grid building effort. In the following, we present an itemized comparison of both frameworks in order to provide the reader a better understanding of the technical aspects involved.

3.1. Vision

JXTA's vision consists of a global, heterogenous and dynamic networked world that conforms to the peer-to-peer (P2P) paradigm. Its goal is to deliver a platform and language neutral base for building an ubiquitous peer-to-peer infrastructure that is able to deliver global connectivity in the presence of networking barriers. It thereby places a strong focus on the dynamic formation of peer communities that span physical networking boundaries. In the JXTA vision, language and platform independence is achieved through reliance on core XML protocols for resource discovery and publication, peer-to-peer communication and peer status monitoring.

Jini's vision entails the creation of a virtually administration free service network that is designed with dynamic service composition and registration in mind. Services may be provided by hardware or software components and are expected to join and leave the

network in a non-orderly fashion. In the Jini vision, clients interact with services through a representative called a proxy. Jini places a strong focus on protocol agnosticism and dynamic installation of client side application logic through the use of mobile code accompanying the proxy.

3.2. Scope

Both platforms share a common ground in that they both provide an infrastructure for *dynamic organization* of networked services through the concept of *groups*, and an infrastructure for *dynamic discovery* and *publication* of networked services. However, the two platforms differ significantly in the application layer they target.

JXTA provides aforementioned infrastructure by deploying a thin virtual communication layer above the network layer provided by the operating system. It thereby creates a virtual network that abstracts away underlying communication mechanisms, network boundaries and operating system differences. The creation of this layer is based on the definition and correct adoption of a number of XML based protocols for message routing and service communication, distributed query processing, resource discovery and dissemination of peer status information. This *technological* achievement forms the core of the JXTA platform.

Jini, on the other hand, was designed to address inherent problems in distributed application development. These include dealing with (a) the dynamic configuration, deployment and maintenance of such systems in the face of network topology changes or service failures, (b) the presence of network latency and overhead, (c) the possibility of partial failures, (d) the need for synchronization among distributed processes, and (e) scalability issues. Jini has addressed these issues by leveraging capabilities of the Java programming language such as OS independence and dynamic class loading. But instead of delivering a pure technical solution, it presents the developer a *distributed programming model*. This programming model integrates the different technical solutions to aforementioned problems in a framework with firmly described semantics. The programming model consists of event, resource allocation and maintenance, transaction, configuration, service invocation, discovery and security models. In short, Jini targets problems in the application layer of distributed system development, while JXTA mainly targets problems in the network connectivity layer of such systems.

3.3. Resource Discovery and Publication

3.3.1. JXTA

All resources in the JXTA network are described by XML documents called *advertisements*. Standard advertisements defined by the JXTA framework include, among others, advertisements for peers, peer groups, pipes and services. JXTA presents general application logic as a resource in the form of module advertisements. Three different classes of module advertisements are defined. *Module Class* advertisements embody a certain behavior, but do not contain the implementation for that behavior nor the means to appeal to it. They can be used to group a set of *Module Specification* advertisements who embody the access specification to the module. In web services terminology, one could compare such an advertisement with a WSDL [16] document. Often such a module specification advertisement contains a pipe advertisement that can be used to set up a communication channel with the service. A third class of module advertisements are the *Module Implementation* advertisements. These contain the physical form of the platform specific application logic for the service that implements a particular module specification. As such, an implementation advertisement may contain source code, a Java jar, a dynamic link library, or a Perl script for example. Traceability through these three levels of abstraction is achieved by embedding the ID of a module class advertisement into the module specification advertisement associated with that class, as well as embedding the ID of such a module specification advertisement in all associated module implementation advertisements.

Advertisements are indexed by one or more of their attributes. The attributes on which an advertisement will be indexed are defined by the advertisement itself. Upon publication of the advertisement, the indices are pushed to stable peers called *rendezvous* peers, who maintain the index tables and respond to queries for advertisements. Every peer in the JXTA

network is permanently connected to one such rendezvous peer. A peer uses a local Single Resource Distributed Index (SRDI) service to compute the indices and push them out to its rendezvous peer. Per attribute, the rendezvous computes a hash in order to determine the rendezvous that will store the index in its table. For every index, the rendezvous' table holds a pointer to the peer hosting the corresponding advertisement.

Figure 1 gives an overview of a typical advertisement lookup scenario. When peer A requests a search for an advertisement, it supplies a key-value pair to its rendezvous (R5). The rendezvous computes the hash for the pair, forwards the request to the mapped rendezvous (R3) who will look up the index in its table. If the index is found, the request is forwarded to the associated peer A who will deliver the advertisement directly to peer B. The requesting peer will cache the delivered advertisement locally. If the index is not found, a limited range walk [17] is issued to propagate the query to other rendezvous. Such a situation may arise when the distributed hashing table is rendered inconsistent by rendezvous leaving the network.

All peers push indices of their local advertisements to their rendezvous in a temporal fashion. As a result, indices of newly discovered advertisements will also be associated with the requesting peer in the rendezvous' tables as illustrated in Figure 2. If a request for the same advertisement arrives at rendezvous R3, it will have the choice of forwarding the request to peer A or peer B, resulting in a form of redundancy in the discovery infrastructure. JXTA also supports the replication of indices among neighboring rendezvous to avoid the need for a limited range walk in case a rendezvous goes down.

In order to prevent the assimilation of stale advertisements in the caches and index tables, advertisements are published with a lifetime which is determined by the peer publishing the advertisement.

3.3.2. Jini

Jini regards a service as an entity in the distributed network which makes its delivered functionality available to clients through the use of a *representative* called a *Remote Proxy* [18]. Such remote proxy objects are registered by the service into registries called *lookup services*. A lookup service (LUS) is always associated with at least one group. Clients discover lookup services by means of multicast or unicast protocols [19]. The result of the discovery

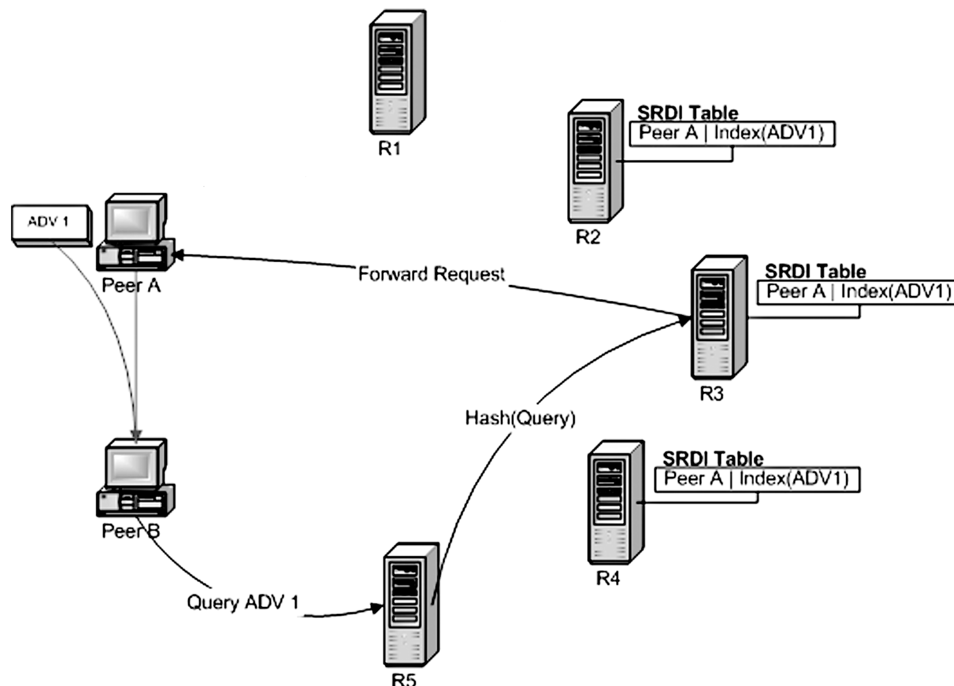

Figure 1. Advertisement query.

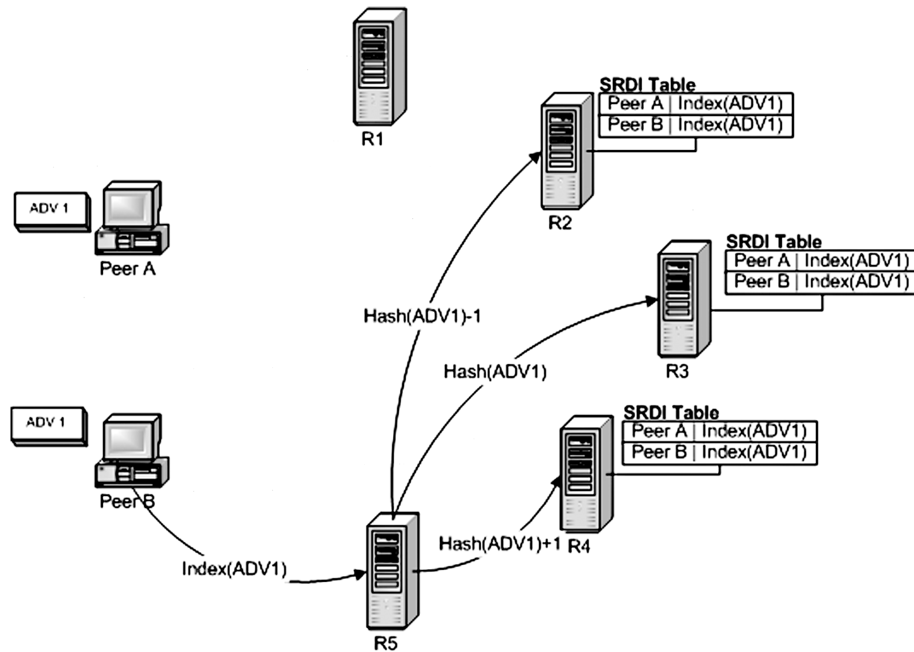

Figure 2. Peer B pushes indices of newly discovered advertisement to its own rendezvous.

process is a LUS proxy delivered to the client. After the LUS proxy is obtained, a client may invoke operations on it to search for service proxies registered in the LUS or to register a proxy for the service hosted by the client. The LUS discovery process may be scoped by indicating the groups of interest.

In order to look up a specific set of service proxies in the LUS, a client submits a *service template* which bundles a number of search criteria. The LUS will return a set of service proxies conforming to the template. A first possible form of proxy lookup consists of *type based* lookup which involves full object-oriented matching semantics. For example, a request for a service proxy implementing interface *A* will match all service proxies implementing interface *A* or subinterfaces of *A*. In general, a proxy matches a lookup request when the proxy object is an instance of all classes specified in the service template.

Apart from service lookup based on Java types, one can also perform *attribute based* lookup. Attributes are defined by the service publisher and are implemented as Java objects. An attribute *A* specified in a service template matches an attribute *B* attached to the proxy, when A and B are of the same class or A's class is a superclass of B's class, and all non-null data fields of attribute *A* equal the corresponding fields in attribute *B*. Data fields are considered equal when they serialize to the same byte sequence.

Lastly, one can also specify the unique ID of the requested service. Such an ID is generated upon the first registration of a service proxy with a LUS, and should be reused by the service host upon subsequent registrations.

Jini adopts a *leasing* [20] model which is also used in the discovery infrastructure. In this model, acquiring a (remote) resource is always paired with negotiating a time period during which this resource will remain allocated by the client. After this period, the resource owner is free to reclaim the resource. A client may lengthen its allocation by renewing the leasing contract. In the context of service discovery and publication, the service host is responsible for renewing *registration* leases. These govern the period of time during which the service proxy will remain registered with the lookup service.

3.3.3. Comparison

In JXTA, discovery requests are routed over the virtual overlay network, thus enabling global, wide-area resource discovery. Jini, on the other hand, only supports ad hoc and zero maintenance resource discovery within the multicast radius of a LAN. To discover resources outside

of the LAN environment, static addresses have to be specified in order to contact associated lookup services. We note that a system administrator may configure a wider discovery scope by configuring the *fiddler* helper service to automatically contact a pre-configured set of external lookup services when a discovery request arrives [21].

Because edge peers cache advertisements themselves, a reliance on rendezvous peers in the JXTA network is avoided. Also, if an edge peer cannot find a rendezvous, it becomes a rendezvous itself. This makes the JXTA discovery network resilient to rendezvous failures. In Jini, reliance on central lookup services is mitigated by the use of IP multicast for LUS discovery. However, because of the limited range of the multicast discovery and the static configuration of lookup service addresses outside of the local LAN, LUS disappearance results in a higher impact on the health of the discovery network in the Jini case.

A limitation of the JXTA discovery infrastructure is the fact that only one name value pair can be used to specify an advertisement query. More general advertisement matching such as the use of regular expressions is not supported.

In Jini, a more semantic and object-oriented form of service lookup is made possible through interface or type based lookup. Although multi-criteria search is supported in Jini, the matching process only supports equality-based matching over service attributes. Jini does deliver a helper utility class called *ServiceDiscoveryManager* to ease the discovery process for the client and perform client side filtering of services returned by the lookup infrastructure. User defined filters can be plugged in to implement more complex service matching semantics. However, implementing an extensive attribute scheme based on Java objects will be more involved compared to the XML based approach used in the JXTA framework.

Compared to Jini, JXTA supports a more layered abstraction for service and module discovery through the notion of module class, specification and implementation advertisements. This is especially useful in order to factor out platform dependencies through the publication of multiple platform dependent and coexisting service specifications and implementations. In the Jini context, the RIO [22] project is aimed at providing more deployment and quality of service oriented support for Jini services through a resource management framework. It offers the possibility to construct *platform*, *behavioral*, and *fault detection* policies to steer the dynamic deployment and maintenance of Jini services.

Both frameworks provide measures to limit the impact of stale resources advertised on the service network. JXTA's model of advertisement lifetime determination by the publisher, is more sensitive to misuse than Jini's model where the lease period is negotiated and can be controlled by the lease grantor.

3.4. Service Interaction

Both frameworks differ in their approach to the distributed nature of networked services. Although a Jini service or proxy may combine functionality of different distributed services in the network, the individual interactions between the entities adhere to the client-server model. In Jini, a service is represented to the client by the use of a proxy implementing a Java interface. This enables the service to rely on the Java type system in order to unambiguously present its service contract to the client, and enforce its compliance.

Proxy code is dynamically downloaded into the client's virtual machine. As a result, extra application logic can be easily embedded in the service proxy and stateful client-service interaction becomes feasible as illustrated in Figure 3. Such a stateful proxy that embeds extra application logic on the client side is called a *smart proxy*. The use of smart proxies fully shields the client from the service implementation and client-to-service communication details. Dynamically downloading the proxy code also eases maintenance of the client side application logic. A change in the central codebase disseminating the proxy code will automatically propagate to all clients.

Jini provides a pluggable service invocation layer called JERI (Jini Extensible Remote method Invocation) with RMI based semantics. Unlike it predecessor, JERI exposes the layers of the service invocation process to the developer. Three layers are defined, the *invocation* layer, the *object identification layer* and the *transport* layer. As a result, new policies and implementations for object marshalling/unmarshalling, distributed garbage collection, wire

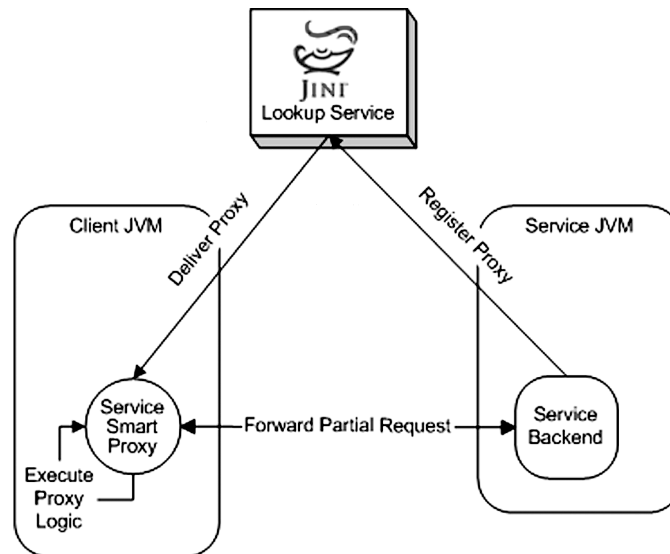

Figure 3. Smart proxies in Jini.

protocols, and method dispatching can be plugged in. Jini's ability to cross networking barriers in the context of service communication is dependent on the transport layer of the JERI stack. Plugging in a transport layer using JXTA pipes would enable such connectivity in an environment with firewalls.

JXTA does not follow such a strict service interaction model. It supports client-server interaction models through *peer services* as well as more collaborative interaction models through *peer group services*. JXTA also adopts a more open view to interface descriptions than Jini. *Module specification* advertisements may be used to define the service interfaces in an arbitrary interface description language. Although client to service communication is also left open ended in the JXTA framework, one should use JXTA pipes for transmitting messages between peers in order to benefit from the global connectivity features of the JXTA overlay network. Messages are to be constructed in XML and are converted to a binary wire format for efficient transfer over the network. XML messages may contain character based data as well as binary data, however it is the developer's responsibility to deal with binary data in a peer environment with a heterogenous platform base.

In the group service model, services are placed on the network in a *declarative* fashion, by simply stating their presence through an advertisement. A client does not have to interact with a representative but can appeal directly to the service network by posting a request that requires collaboration among different service instances. The key service within the JXTA framework that enables this form of service interaction, is the *resolver* service. The resolver enables nodes within the network to automatically distribute generic queries among service instances and receive the associated responses. In doing so, it delivers no guarantees regarding the reliability of request or response delivery.

Services may be pre-installed on peers or dynamically loaded into peers by searching for the appropriate module implementation advertisement and instantiating that module on the peer. This way, peers can dynamically contribute to build a fault tolerant service network by instantiating the group services of interest.

We conclude that Jini offers a deterministic service invocation model. A service is presented by a single access point, embodied in the service proxy, that carries the full responsibility for handling all client-to-service interactions and ensuring the quality of service requirements are met. Jini thereby states that such handling should not be statically laid down. Therefore it requires an environment that enables objects, but also object code, to move around the network. This enables the dynamic change of service interaction handling policies, by updating the service proxy within the lookup services. JXTA does not prescribe the Java type system for interface specification and also supports non-deterministic and more collaborative forms of service interaction, independent of networking barriers.

3.5. Group Concept

There is a profound difference in the practical manifestation of the grouping concept in both frameworks. Although Jini places a conceptual focus on the notion of communities and federations, the actual manifestation of this group concept in the framework APIs is rather limited. Groups only fulfill a small scoping role during the LUS discovery phase. In this phase, a client may constrain the set of returned lookup services by specifying a number of group names. If a LUS supports one of the mentioned groups, its proxy is returned to the client. After discovery of the LUS, service lookup or registration may take place. During these phases, groups are no longer taken into account. More specifically, when a LUS L1 supports groups A and B, then all service publication and lookup requests will be performed within the context of group A as well as group B. As a result, a service may be registered with L1 which was discovered by specification of group A, yet its proxy will also match requests for service lookup by clients who discovered L1 through specification of group B.

The peer group has a more central and important role within the JXTA framework. Every peer group defines its own group infrastructure components. These include, among others, components for advertisement discovery, group membership, query propagation, routing and peer monitoring. The peer group defines a number of standard slots for core group infrastructure which are to be filled in with the identifiers of the associated module advertisements delivering these components. As a consequence of this logical and structural partitioning, only peers that are member of the group are able to use the group infrastructure and discover resources within the group. JXTA allows for groups to be hierarchically structured, a standard top level group called the *World* group delivers the minimal infrastructure to discover and become part of other subgroups.

3.6. Security

Jini offers a security model that includes support for mutual client-server authentication and authorization, safe delivery of mobile code and secure client-server communication. Security constraints are placed on the proxy, as all client to service interactions take place through the service proxy. Jini supports mutual authorization, allowing both server and client to place constraints on the service proxy. An example of such a constraint is to require the client to authenticate itself. For authentication and authorization, Jini builds further on the abstractions defined by the Java Authentication and Authorization Service (JAAS). JAAS supports different identity models and authentication schemes such as X.500 or Kerberos.

As proxy code is downloaded dynamically into the client's virtual machine, it is important to place trust in and guarantee integrity of the service proxy. To achieve proxy trust, the client will request a bootstrap proxy, which consists of local trusted code, to provide a *ProxyVerifier* who will determine if the downloaded proxy is to be trusted by the client. Security constraints can be set on the proxy with method level granularity. Such method level constraints are integrated in the service invocation layer through the JERI infrastructure.

Jini supports both integrity of in-band data exchanged during client to service communication, as well as the integrity of mobile code which is delivered out of band. To guarantee mobile code integrity and achieve proxy integrity, a URL scheme called HTTPMD is used. This scheme annotates a codebase URL which is attached to the marshalled proxy, with a hash of the jar containing the code. Upon download, a hash of the downloaded code is automatically computed and compared to the hash specified in the codebase URL. Jini supports different algorithms to be plugged in to perform integrity validation.

JXTA does not advocate a particular security model [23]. It provides hooks for integrating existing security infrastructures into the peer group. For secure pipe communication, JXTA supports pipes conforming to the Transport Layer Security (TLS) standard. TLS requires peer certificates to set up a secure communication channel. The dissemination of such certificates, however, is not standardized in the JXTA framework. In a trivial setting, every peer acts as its own certification authority. More advanced peer group infrastructures may ship with their own certification authority (CA). In JXTA, the peer group may be used as a security context. Access to the peer group, and by consequence to all services advertised in the group, can be guarded by a group membership service. Such a membership

service may require a peer to authenticate itself before access to the group and its services is granted. JXTA also ships with a security toolkit which delivers a number of cipher and hashing algorithms.

3.7. Flexibility

The Jini framework is entirely Java based and the realization of its philosophy relies on the capabilities of the Java programming language. Although Java can guarantee platform independence, it implies language dependence. Every non-Java entity that wants to participate in a Jini network, needs a Java representative within the network. Since Jini binds developers to the Java language, it also requires network participants to run a Java virtual machine and to have access to a number of standard Java libraries.

Since JXTA is based on XML, it shows no platform or language dependencies. To participate in the JXTA network, the only requirements are the presence of an XML parser and some arbitrary form of network connectivity. As mentioned in Section 3.5, core infrastructure components can be replaced by peer group specific implementations.

A point of inflexibility that is present in both platforms regards their data model. In Jini one has to adhere to the Java object model, in JXTA all messages and advertisements should be embedded in XML. This is a direct consequence of the aforementioned language bindings.

3.8. Ease-of-Use

The inclusion of a number of well thought out high-level helper utilities, form a solid base that facilitates the implementation of Jini services. These utilities lower the cost of programming in the Jini model considerably. Nearly all bookkeeping tasks that need to be done by an entity participating in the Jini network, can be delegated to these utilities. Helper services are provided to ease the tasks such as service discovery and publishing, lease management and the construction of secure proxies. JXTA does not yet provide such a broad range of utilities. It is the communities' responsibility to develop high-level services on top of the JXTA platform that could further ease or accelerate development using JXTA.

4. CONTRIBUTIONS OF JINI AND JXTA TO AN LWG MIDDLEWARE

In this section we discuss a mapping of the features outlined the previous section, to the middleware requirements of a lightweight Grid as discussed in section two.

4.1. Limited Administrative Resources

The creation of *virtually administration free* service communities is one of the key mission statements of the Jini framework. The most notable contributions of Jini in this regard are:

- Proxy autonomy: The use of proxies allows the service designer to include client side logic that automates certain tasks and eases the amount of administration forced upon the user. An example is to include a user interface that is automatically deployed on the client machine.
- Low cost service updates: Services are easy to upgrade. As soon as a new version of the service proxy is uploaded and the service provider discontinues the lease on the old proxy, client side discovery will result in the new service proxy being returned. As code is automatically downloaded to the client, no manual steps are involved to upgrade the client side proxy code.
- Dynamic registration/deregistration: As soon as services go live they automatically register with the discovery infrastructure. When the service's hosting environment goes down the registration lease will be discontinued and all stale information is automatically cleared from the discovery infrastructure. Jini thus provides low cost administration on the discovery infrastructure with automated clearance of stale resources.

- Low cost middleware installation: Java based middleware can be installed across the world wide web with a single click through the use of Java Webstart or custom applets.
- Leasing Model: The leasing model as a whole contributes to the development of autonomous self healing services that are able to recover from crashes and clear stale information without administrator intervention.

4.2. Heterogeneity

As both Jini and JXTA have Java based implementations, the features of the Java language for dealing with OS and hardware heterogeneity are inherited by both frameworks. These features include uniform access to the file system and the network, code and data portability across different architectures and a uniform security framework. We note however that a pure Java based approach is not mandated by Jini nor JXTA. While Jini requires a Java platform on the client side to cope with system heterogeneity, a service's backend can be implemented in any language on any platform. JXTA on the other hand, aims to provide different implementations of its framework for different languages and systems. These implementations are able to communicate with each other because of the framework's standardized XML protocols. Consequently, both frameworks are able to integrate non-Java services in the Grid middleware. Also, support for heterogeneous networks is one of JXTA's key goals. The presence of relays in the network is transparently used by the JXTA routing protocols to provide communication links between peers across protocol and firewall boundaries.

Despite these contributions, significant heterogeneity issues still need to be dealt with by the middleware itself. These include the dependencies Grid applications will have on a particular execution environment, extension of scheduling algorithms to cope with widely varying system characteristics and the consequences of site autonomy in respect to local security policies.

4.3. Volatility

The idea that distributed resources are volatile is inherently addressed in the Jini philosophy. Features such as dynamic registration and deregistration of services in the discovery architecture with proper client notification, support for two phase commit transactions, a leasing model, and the presence of sequence numbers in the eventing system reflect this. In Jini, every client interacts with a service's backend through a mediator called a proxy. The use of smart proxies enables service developers to include fault recovery policies in the proxy in order to shield clients from the consequences of a service going down. A Jini proxy thus contributes to the fulfillment of the *failure transparency* requirement. Jini services also integrate well with the RMI activation framework which enables the automatic recovery of services in the event of the hosting JVM going down and coming back up again. This fulfills the *persistence transparency* requirement. Jini's strong support for mobile code and mediation of communication channels through smart proxies, contribute to the *migration transparency* that is necessary to adapt to changing fabric conditions in a volatile environment. Jini deals with the *transaction transparency* requirement by offering standard services for setting up two phase commit transactions.

JXTA's *group service* model automatically distributes queries among all service instances in the group, and thus provides a fault tolerant way of accessing service functionality. As long as one service instance is up, the query will be answered. JXTA also features a fault tolerant message routing infrastructure. Upon detection of a failed link a new route is automatically resolved for the message. JXTA uses a distributed hash table (DHT) [17] approach to provide efficient discovery of resources and avoid flooding the network with discovery queries. However, maintaining the consistency of a DHT under volatile conditions is costly. JXTA combines a loosely consistent DHT with limited range walks around the network to deal with this trade-off. It mitigates the cost of keeping a DHT consistent under high volatility rates and can deal with varying levels of node volatility.

4.4. Scale

Jini has firm roots in object-oriented distributed programming. Its modular architectural style and programming model contribute to the development of extensible middleware that supports customized deployment depending on the scale of deployment. This is demonstrated by the Jini ICENI project (see Section 5). Jini lookup services have been found to scale well as the number of registered services increases [24]. Scalability studies of the same extent are not available for JXTA although performance studies of the framework on a smaller deployment scale are available [25, 26].

Jini lacks support for discovering computational resources on a global scale in a fully distributed manner, but JXTA does have this capability. Integrating Jini and JXTA discovery infrastructures and plugging in JXTA pipes into the JERI protocol stack is a possible course of action in this regard. Other schemes for extending Jini's discovery reach exist [27–29]. Ubiquitous large scale discovery and communication is one of JXTA's key goals. A fully distributed discovery infrastructure based on a hierarchical peer-to-peer network, and the ability to assign infrastructure resources such as routing, discovery and membership services to specific peer groups, contribute to the fulfillment of the scalability requirement.

In principle, JXTA's peer-to-peer infrastructure is better qualified to address scalability requirements by avoiding central server bottlenecks on a network and system load level. However, there have been practical problems in the past regarding the scalability of JXTA's implementation. Although the recent 2.0 release specifically addressed scalability issues, more research is necessary to quantitatively investigate JXTA's scalability.

4.5. Openness

Jini and JXTA do not provide features that have a direct impact on the openness of the resulting LWG middleware. As mentioned in Section 2, making a computing platform openly accessible involves creating incentive for resource owners to share and consumers to consume, which is beyond the scope of both frameworks. However, JXTA's pluggable approach to group membership services and the ability of every peer to start, join or leave a peer group, does support the creation of more dynamic and open communities than currently found in heavyweight Grid VOs.

Jini's support for mobile code is beneficial to the development of an agent based framework that is able to implement a dynamic form of service level agreement (SLA) negotiation between providers and consumers. In this regard, group communication in a JXTA VO can also help in implementing economic pricing and bidding models.

4.6. Hostility

Both Jini and JXTA offer means to protect information exchange in a hostile environment. Jini provides TLS/SSL and Kerberos GSS-API based communicators for client service interactions, while JXTA provides TLS pipes for secure communication. Jini also has strong support for safe delivery of mobile code.

Mutual client-server authentication and authorization schemes integrated within the Jini framework, allow involved parties to establish identity, trust and grant each other the right access permissions. Once parties trust each other enforcement of access rights is further controlled by the security provisions delivered by the Java platform. The security subsystem of a JVM can be completely tuned using *policy files*. They specify what code can perform which operations. This includes sandboxing the mobile code to limit its range of impact on the hosting system. Although this scheme allows for safe execution of pure Java code, it has no support for safe execution of non-Java binaries. If, through the use of JNI (Java Native Interface) or the creation of new subprocesses, native code is executed, it will execute as originating from the owner of the JVM process and it is up to the operating system to determine its permissions.

Apart from TLS transports, JXTA also includes standardized slots for group membership services that control peer access to the peer group in question, a feature missing in Jini. The implementation of this group service is up to the peer group creator. For lightweight Grid

deployments, a group membership model that is based on peer reviewing [30, 31] would be an interesting option for certain virtual organizations.

5. OVERVIEW OF JINI AND JXTA IN LIGHTWEIGHT GRID SYSTEMS

In this section we present an overview of a number of lightweight Grid systems that incorporate Jini or JXTA. We do not make a comparison between these systems, but focus exclusively on the role that Jini or JXTA plays in the system.

5.1. LWGs and Jini

5.1.1. ALiCE

ALiCE (Adaptive and scaLable internet-based Computing Engine) [32] is a Java based Grid computing middleware. It supports the development and deployment of generic Grid applications on heterogeneous and non-dedicated resources. The ALiCE system consists of a core middleware layer that manages the Grid fabric. It hosts compute, data and security services, as well as monitoring and accounting services. The core layer also includes an Object Network Transport Architecture (ONTA) component that deals with the transportation of objects and associated code across the Grid fabric. The core layer uses Jini technology to support the dynamic discovery of resources within the Grid fabric. For object movement and communications within the Grid, a Jini based tuple space implementation called JavaSpaces [33] is used. A commercial implementation of the JavaSpaces platform, called GigaSpaces [34], is also supported.

5.1.2. JGrid

JGrid is a Jini based Grid project specifically aimed at exploiting Jini's support for dynamic self healing systems. To virtualize a local computing resource, *Compute Services* are introduced that manage job execution on the local JVM. These Compute Services can be hierarchically grouped using *Compute Service Managers*. Resource brokers schedule jobs onto these Compute Service Managers and Compute Services on a client's behalf.

JGrid supports a number of application models including batch style execution using a Sun Grid Engine backend, MPI (Message Passing Interface) [35] enabled execution and the farmer worker execution pattern. JGrid has extended Jini's discovery infrastructure to cope with wide area discovery [36]. Lookup services are connected together in a k-ary tree topology that reflects the associated geographical topology. Lookup services at the leaves have an organizational unit scope, while services at the top have continental scope. The discovery architecture as a whole is made robust by adding support for lookup service replication at every level in the tree. Parallel program development is to be supported by a graphical application development environment called P-Grade [37].

5.1.3. ICENI

The Imperial College e-Science Networked Infrastructure (ICENI) [38] is a service oriented Grid middleware written in Java. It provides a component oriented programming model to the Grid in which users define components on an abstract level. Administrators and users are able to provide different concrete implementations for these abstract components.

Components can be selected and linked together in a graphical DAG editor. This editor enables the users to define an abstract *Execution Plan* (EP) based on the high level actions taken by the workflow. The components in the EP are described in terms of meaning (high level description), behavior (control flow through the component) and implementation (algorithm used, I/O ports, data-types used by the component). A scheduler takes the EP and searches for concrete implementations based on the available resources. *Launching Services* running on the computational resources map jobs to a native format that is used by the local OS or a distributed resource management system such as Condor. ICENI has launchers for fork, Condor, Globus, and Sun Grid Engine. The launching framework is also

responsible for staging the necessary input and output files as well as monitoring running jobs and delivering performance data.

ICENI uses Jini as a basis for its service infrastructure. However, gateways are provided in order to make ICENI components available externally through interfaces that are compliant with the OpenGrid Services Architecture (OGSA) [39].

5.1.4. JISGA

The Jini Service-Based Grid Architecture (JISGA) [40] is a workflow oriented Grid architecture based on Jini. All middleware components of the JISGA system are implemented as Jini services. To enhance interoperability with external partners and open up the architecture to current standardization efforts, JISGA services are also made available as Web Services.

JISGA supports an XML based job and workflow description language called SWFL (Service Workflow Language) which is an extension of WSFL. A JISGA job consists of a composition of services in the system. A workflow engine automatically maps the workflow to the underlying service oriented infrastructure.

JISGA also uses JavaSpaces as a distributed shared memory mechanism. JavaSpaces are used for storing job queues and as a general communication medium between the distributed processes in the system.

5.2. LWGs and JXTA

5.2.1. The Compute Power Market

The Compute Power Market (CPM) [8] project focuses on the installment of market-based approaches in global resource sharing environments. The market's scheduler supports deadline or cost-driven scheduling policies while its accounting component hosts the associated accounting and billing functionality. CPM envisages a market wherein both supercomputing centers and desktop owners share resources such as bandwidth, CPU cycles and storage space. An economical market based approach and an associated accounting infrastructure are deemed necessary in order to provide incentive for resource owners to bring their resources into the market. CPM uses JXTA pipes for secure communication between the different entities in the market. Resource providers, consumers and markets find each other through the JXTA discovery infrastructure.

5.2.2. Triana

Triana is a component based problem solving environment written in Java [41]. The environment ships with a GUI for workflow editing and support for exporting workflows to various formats (BPEL4WS, WSFL, Petrinet). Triana services take part in a workflow through their input and output ports. These services can be grouped into a *Group Unit* that is coordinated by a *Control Unit*. The Control Unit supports a peer-to-peer and a task farming model for the distribution of the Triana services in a group across the Grid fabric.

The higher level Triana components program to the Grid Application Toolkit (GAT) interface in order to shield them from issues specific to the distributed middleware used. The GAT provides a high level application-driven API that is middleware independent. It contains the functionality application programmers need from Grid Services. Triana currently has GAT bindings for JXTA and Globus. In the JXTA binding, service input and output ports are implemented as JXTA pipes. The GAT ID of the service is mapped to a JXTA ID and discovery of these services is done using the JXTA overlay network.

6. CONCLUSION

In this contribution we have looked at the key characteristics of the environment in which a lightweight Grid has to operate and how this determines requirements for lightweight Grid middleware. We have presented a technical overview of Jini and JXTA in order to analyze their possible contributions to the fulfillment of these requirements. Although neither platform delivers a silver bullet to the Grid middleware developer, we have identified areas in which both platforms clearly can contribute. Both technologies can play a role in constructing

a lightweight Grid middleware that will provide a more open and dynamic base for deploying Grids in a hostile, volatile and heterogeneous environment with limited administrative resources.

REFERENCES

1. The Search for Extraterrestrial Intelligence project, http://setiathome.ssl.berkeley.edu/.
2. I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations,” International Journal of High Performance Computing Applications 15, 200–222 (2001).
3. I. Foster and C. Kesselman, “The Grid 2: Blueprint for a New Computing Infrastructure.” Morgan Kaufmann, 2003.
4. F. Berman, G. Fox, and A. Hey, “Grid Computing: Making the Global Infrastructure a Reality.” John Wiley & Sons, 2003.
5. Sun Microsystems, Inc., “Jini Project Home Page,” http://www.jini.org, 2004.
6. Sun Microsystems, Inc., “JXTA Project Home Page,” http://www.jxta.org, 2004.
7. J. Chin and P. Coveney, Towards Tractable Toolkits for the Grid: A Plea for Lightweight, Usable Middleware, UK e-Science Technical Report, number UKeS-2004-01, 2004.
8. R. Buyya and S. Vazhkudai, “Compute Power Market: Towards a Market-Oriented Grid,” In Proceedings of the first International IEEE Symposium on Cluster Computing and the Grid, Brisbane, Australia, 2001, pp. 574–581.
9. T. Ping, G. Sodhy, C. Yong, F. Haron, and R. Buyya, In “A Market-Based Scheduler for JXTA-Based Peer-to-Peer Computing System,” Springer-Verlag Lecture Notes in Computer Science Series 3046, 2004, pp. 147–157.
10. D. Abramson, R. Buyya, and J. Giddy, “A computational economy for Grid computing and its implementation in the Nimrod-G resource broker,” Future Generation Computer Systems 18, 1061–1074 (2002).
11. G.Fox and D. Walker, “e-Science Gap Analysis,” http://www.nesc.ac.uk/technical_papers/UKeS-2003-01/GapAnalysis30June03.pdf
12. W. Edwards, “Core Jini 2nd Edn.,” Prentice Hall PTR, 2000.
13. D. Brookshier, D. Govoni, N. Krishnan, and J. C. Soto, “JXTA: Java P2P Programming,” Indiana, Sams Publishing, 2002.
14. P. Deutsch, “The eight fallacies of distributed computing,” http://today.java.net/jag/Fallacies.html, 2004.
15. B. Traversat, A. Arora, and M. Abdelaziz, “Project JXTA 2.0 Super-Peer Virtual Network,” http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf, Sun Microsystems, Inc., 2003.
16. R. Chinnici, M. Gudgin, J. Moreau, J. Schlimmer, and S. Weerawarana, “Web Services Description Language (WSDL) 2.0,” http://www.w3.org/TR/wsdl20.html, 2004.
17. E. Pouyoul, B. Traversat, and M. Abdelaziz, “Project JXTA: A Loosely-Consistent DHT Rendezvous Walker,” http://www.jxta.org/project/www/docs/jxta-dht.pdf, Sun Microsystems, Inc., 2003.
18. R. Johnson, J. Vlissedes, E. Gamma, and R. Helm, “Design Patterns—Elements of Reusable Object Oriented Software.” Addison Wesley, Reading, MA, 1995.
19. Sun Microsystems, Inc., “Jini Technology Core Platform Specification,” Sun Microsystems, Inc., 2003.
20. M. Kircher and P. Jain, “Leasing pattern,” In Proceedings of Pattern Language of Programs conference, Allerton Park, Monticello, Illinois, USA, 1996.
21. S. Li, R. Ashri, M. Buurmeijer, E. Hol, B. Flenner, J. Scheuring, and A. Schneider, “Professional Jini.” Birmingham, Wrox Press, 2000.
22. D. Reedy, “Project Rio A Dynamic Adaptive Network Architecture,” Sun Microsystems, Inc., 2003.
23. Sun Microsystems, Inc., “Security and Project JXTA,” Sun Microsystems, Inc., 2002.
24. M. Kahn and C. Cicalese, “CoABS Grid Scalability Experiments,” Journal of Autonomous Agents and Multi-Agent Systems 7, 171–178 (2003).
25. E. Halepovic and R. Deters, “The Costs of Using JXTA,” In Proceedings of the 3th IEEE International Conference on Peer-to-Peer Computing (P2P’03), Linköping, Sweden, 2003, pp. 160–167.
26. E. Halepovic and R. Deters, “JXTA Performance Study,” In Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM’03), Victoria, BC, Canada, 2003, pp. 149–154.
27. Z. Juhasz, A. Andics, and S. Pota, “JM: A Jini Framework for Global Computing,” In Proceedings of the 2nd International Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems at IEEE International Symposium on Cluster Computing and the Grid (CCGrid’2002), Berlin, Germany, 2002, pp. 395–400.
28. T. Hodes, S. Czerwinski, B. Zhao, A. Joseph, and R. Katz, “An Architecture for Secure Wide-Area Service Discovery,” Wireless Networks, 8, 213–230 (2002).
29. W. Tseng and H. Mei, “Inter-Cluster Service Lookup Based on Jini,” In Proceedings of the 17th IEEE International Conference on Advanced Information Networking and Applications, Xian, China, 2003, pp. 84–89.
30. R. Chen and W. Yeager, “Poblano, a Distributed Trust Model for Peer-to-Peer Networks,” Technical Report, Sun Microsystems Inc., http://www.jxta.org/project/www/docs/trust.pdf, 2001.
31. S. Kamvar, M. Schlosser, and H. Molina, “The EigenTrust Algorithm for Reputation Management in P2P Networks,” In Proceedings of the 12th International World Wide Web Conference, Budapest, Hungary, May 2003, pp. 640–651.
32. Y. M. Teo and X. B. Wang, “ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing,” In Proceedings of IFIP International Conference on Network and Parallel Computing, in Springer-Verlag, Lecture Notes in Computer Science Series 3222, Xian, China, 2004, pp. 101–109.

33. E. Freeman, S. Hupher, and K. Arnold, "JavaSpaces Principles, Patterns, and Practices." Addison-Wesley, New York, 1999.

34. The GigaSpaces homepage, http://www.gigaspaces.com/.

35. W. Gropp, E. Lusk, and A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface." Cambridge, MA, MIT Press, 1999.

36. Z. Juhasz, A. Andics, K. Kuntner, and S. Pota, "Towards a Robust and Fault-Tolerant Multicast Discovery Architecture for Global Computing Grids," In Proceedings of the 4th DAPSYS workshop, Linz, Austria, 2002, pp. 74–81.

37. C. Nemeth, G. Dozsa, R. Lovas, and P. Kacsuk, "The P-GRADE Grid Portal," In Proceedings of the 2004 International Conference on Computational Science and its Applications, Assisi, Italy, 2004, pp. 10–19.

38. N. Furmento, J. Hau, W. Lee, S. Newhouse, and J. Darlington, "Implementations of a Service-Oriented Architecture on top of Jini, JXTA and OGSI," In Proceedings of the Second Across Grids Conference, Nicosia, Cyprus, 2004.

39. J. Joseph and C. Fellenstein, "Grid Computing." Prentice Hall PTR, 2003.

40. Yan Huang, "JISGA: A Jini-Based Service-Oriented Grid Architecture," International Journal of High Performance Computing Applications 17, 317–327 (2003).

41. S. Majithia, M. Shields, I. Taylor and I. Wang, "Triana: A Graphical Web Service Composition and Execution Toolkit," In Proceedings of the IEEE International Conference on Web Services (ICWS'2004), San Diego, California, USA, 2004, pp. 514–521.