

# JINI: A Technology for 21<sup>st</sup> Century -- Is it Ready For Prime Time?\*

**Prof. Steven A. Demurjian, Sr.**  
Computer Science & Engineering Dept.  
The University of Connecticut  
Storrs, CT 06269-3155  
steve@enr.uconn.edu  
Tel: 860.486.4818  
Fax: 860.486.4817

**Dr. Paul Barr**  
The MITRE Corp  
145 Wyckoff Road  
Eatontown, New Jersey 07724  
poobarr@mitre.org  
Tel: 732-935-5584  
Fax: 732-544-8317

## 1. Introduction and Motivation

Distributed computing applications for the 21st century are network centric, operating in a dynamic environment where clients, servers, and the network itself all have the potential to change drastically over time. A *distributed application*, a system of systems, must be constructed, consisting of legacy, commercial-off-the-shelf (COTS), database, and new client/server applications that must interact to communicate and exchange information between users, and allow users to accomplish their tasks in a productive manner. The issue is to promote the use of existing applications in new and innovative ways in a distributed environment that adds value. To adequately support this process, the network and its software infrastructure must be an active participant in the interoperation of distributed applications. Ideally, we are interested in distributed applications that plug-and-play, allowing us to plug in (and subtract) new “components” as needs, requirements, and even network topologies change over time.

JINI [Arno99, JINI, JINIARCH] is a new architecture built on top of Java’s remote method invocation (RMI) that promotes the construction and deployment of robust and scalable distributed applications in a network centric setting. JINI technology is forcing software designers and engineers to abandon the client/server view in order to adopt a *client/services* view. In JINI, a distributed application is conceptualized as a set of services (of all resources) being made available for *discovery* and use by clients. To accomplish this, JINI makes use of a *lookup service*, which is essentially a registry for tracking the services that are available within a distributed environment. Services in JINI *discover* and then *join* the lookup service, registering the services (of each resource) that are to be made available on the network. Thus, JINI is conceptually very similar to a distributed operating system, in the sense that resources of JINI are very similar to OS resources. However, in JINI these resources can be dynamically defined and changed. To illustrate JINI, consider that a service `register_for_course(course#)` for a Course database in a University application may be registered with the lookup service. Clients request services by interacting with the lookup service, e.g., asking for `register_for_course(CSE230)`. The lookup service returns a proxy to the client for the location of the service. The client then interacts directly with the service via the proxy to execute the service, e.g., registering for CSE230. In this process, there are a number of important observations. First, services can come (register and join) and go (leave) without impunity, since all interaction with services occurs via the lookup service. Second, clients locate and utilize services without knowing their location on the network, allowing clients to work without interruption as long as “some” service can be located to meet their needs. Third, the location of clients and/or services on the network can change at any time without impacting the network or the users.

Our efforts are motivated from two perspectives. First, by Army requirements, we evaluated the JINI technology in support of present and future systems. Second, as part of grant from AFOSR on large-scale, multi-agent, distributed mission planning and execution in complex dynamic environments, we have been considering the ability of software agents (written using Java) to interact with JINI resources and services. In both efforts, there are a number of common, fundamental questions:

- **Can JINI Support Highly-Available Distributed Applications?**
- **Can JINI Support an Environment with Dynamic Clients and Replicated Services?**
- **Will Clients Continue to Operate Effectively if Replicated Services Fail?**
- **Can JINI be Utilized to Maintain “minutes-off” Data Consistency of Replicas?**
- **Is JINI Easy to Learn and Use? What is Maturity Level of JINI Technology?**

\* The work in this paper has been partially supported by a contract from the Mitre Corporation (Eatontown, NJ) and AFOSR research grant F49620-99-1-0244.

## JINI: A Technology for 21<sup>st</sup> Century?

The reality is that new technologies offer new challenges, with the potential to reap benefits if adopted. However, for future Army systems, it is important that a careful balance is drawn to opt for mature technologies while targeting emerging technologies with potential. The key issue is where JINI fits – as a mature technology or yet another one with potential? The remainder of this abstract reviews JINI, our experimental prototyping effort, summarizes our results, and proposes a series of future work to answer the question: “**Is JINI Ready for Prime Time?**”

### 2. JINI

Stakeholders (software architects, designers, and implementors) can utilize JINI to construct a distributed application by federating groups of users (clients) and the resources that they require. In JINI, the resources register *services* which represent the functions that are provided for use by clients (and other services). In a sense, the services are similar in concept to public methods that are exported for usage as part of an applications class library (API). JINI is versatile, and allows a service to represent any entity that can be used by a person, program (client), or another service, including: a computation, a persistent store, a communication channel, a software filter, a real-time data source (e.g., sensor or probe), a hardware device (e.g., printer, display, etc.), and so on. The services are registered with a *look-up service*. The registration of services occurs using a *leasing* mechanism. With leasing, the services of a resource can be registered with the lookup service for a fixed time period or forever (no expiration). The lease must be renewed by the resource prior to its expiration, or the service will become unavailable. This feature, in part, supports high availability, since it requires the resources to constantly reregister their services; if a resource goes down and does not reregister, the leases on its services expire, and the services will then be unavailable from the lookup service.

As a technology, JINI provides an infrastructure to design and construct distributed applications with a network centric approach that assumes an environment where there is a requirement for the spontaneous interaction of clients and services. Spontaneity from a client perspective supports the dynamic behavior of clients, where they enter and leave the network unpredictably. While connected, clients are guaranteed that either the visible services are available or that failure can be trapped and handled. Spontaneity from a resources perspective, means that when resources fail, the network can adapt, to insure that redundant services, if available, are now accessible to clients. Operationally, when a client wishes to interact with a service, the interaction can occur by either a download of code from service to client, or the passing of a proxy which allows a RMI-like call by the client to the service.

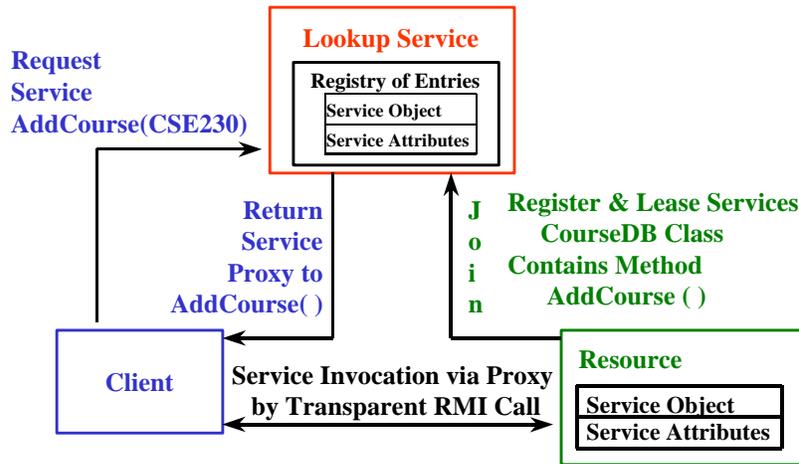
The lookup service is the clearinghouse of a JINI network centric application, since all interactions by resources (e.g., discovering lookup services, registering services, renewing leases, etc.) and by clients (e.g., discovering lookup services, searching for services, service invocation, etc.) must occur through the lookup service. When there are multiple lookup services running on a network, it is the responsibility of the resources to register with them (if relevant). Clients can interact with multiple lookup services, and in fact, it is possible for groups of clients to be established that will always consult a particular “close” lookup service, dictated perhaps by network topology or traffic. Whenever resources leave the environment (either gracefully or due to failure), the lookup service must adjust its registry. There is a time lag between the resource leaving and the removal of services from the registry. Clients must be sophisticated enough to be able to dynamically adjust to these situations.

After discovery has occurred, the resources register services on a class-by-class basis. The class is registered as a *service object* which contains a Java programming interface to the service, namely, the public methods available to clients coupled with a set of optional descriptive service attributes. This registration process is referred to *joining* and is shown in Figure 1. In JINI terms, the service object is registered as a *proxy*, which contains all of the information that is needed to invoke the service. In the request for service, shown in Figure 1, a client will ask for the service to register for a course of the CourseDB class based on the signature of the method: `status register_for_course(int)`. The lookup service will return a service proxy that allows the client to invoke any or all of the methods defined within the service. Using the proxy, the client invokes the needed method(s) as it would any other Java method; the call transparently utilizes RMI with the result of the call returned to the client. The interaction between the client and the resource occur independent from the lookup service.

A lease is the part of the JINI programming model that allows the resources to set the limits of its utilization of services, and allows the lookup service to remove services from its registry that are no longer available. A resource can lease a service to a lookup service forever (not recommended) or lease with a specific expiration date (in milliseconds). If leased using an expiration date, the resource is responsible for renewing the lease prior to its expiration. The leasing and renewal process is intended to keep the registry fresh, containing all active and working

## JINI: A Technology for 21<sup>st</sup> Century?

services. This is of particular importance in a distributed application where resources leave the network due to failure or other reasons. When a resource leases its services with specific expiration times, if failure occurs, when the lease expires and is not renewed, the services will no longer be available. In addition, the lookup service periodically checks to see if services (and resources) are active. Whenever failure occurs, there is a time period when services will be listed in the registry that are unavailable to clients, and in fact clients will receive exceptions if they try to execute such services. Thus, even if a client receives the proxy for a service that is active in the registry, there is no guarantee that the service will be available when invoked. Thus, it is imperative that software engineers design clients that are able to handle this situation.



1. Client Invokes AddCourse(CSE230) on Resource
2. Resource Returns Status of Invocation

**Figure 1: Join, Lookup, and Invocation of Service.**

### 3. Experimental Prototyping Effort

We have taken an experimental prototype approach to evaluate the capabilities of JINI under WinNT to determine if JINI is “ready for prime time”. The goal of the experimentation is to explore the ability of JINI to support applications that require high availability (via replication of resources and their services and data) in an environment where the replicated resources are volatile. Clients, which are also entering and leaving the network, consult the JINI lookup service to locate and subsequently execute the “services” of the replicated resource that are necessary to carry out their respective tasks. If one of the services fails, there is a back-up service that can be utilized to support the client. The replicated databases must be kept consistent, but at any given time point, the data in one database might be “minutes off” the data in the other databases. Over time the databases will synchronize and contain the same information. It is crucial that updates not be lost during the modification and synchronization processes.

A total of six experimental prototypes have been developed modeled on a university application where Persons (students and faculty) are attempting to access and/or modify information related to a course schedule. Students and faculty have a GUI (Java client application) through which they must enter their name and password, and once verified, are able to access course information. To support this, both a PersonDB (for authentication and authorization) and a CourseDB must be available. These two databases are stored in Microsoft Access, and a Java application or database resource, offers a set of “services” that are made available by registration with JINI to clients. A Java GUI client consults the JINI lookup service to search for appropriate services of the replicated database resource that can satisfy their requirements as needed by the student/faculty request. Whenever a Java GUI client modifies the CourseDB as a result of a user request, all other replicated CourseDBs must be modified so that the replicas remain consistent. However, there may be a time difference where the data in one CourseDB is minutes off the data in the other CourseDBs. For discussion purposes, Prototype 6 is shown in Figures 2 and 3.

## JINI: A Technology for 21<sup>st</sup> Century?

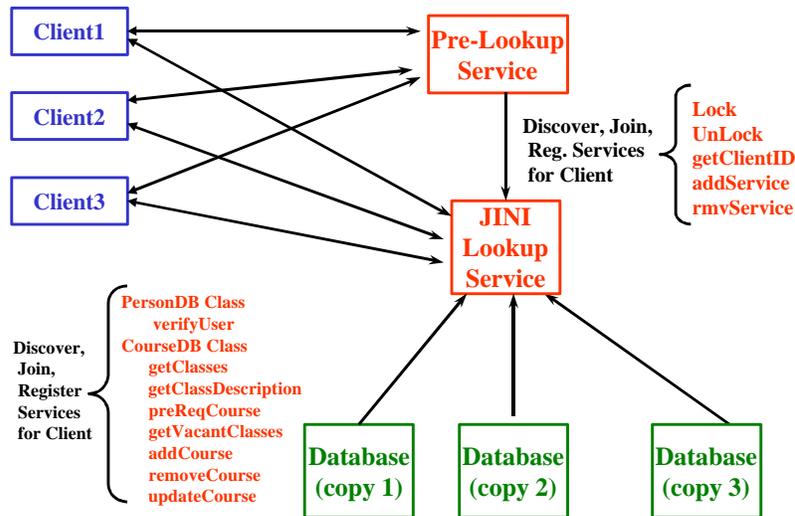


Figure 2: Pre-Lookup Services in Prototype 6.

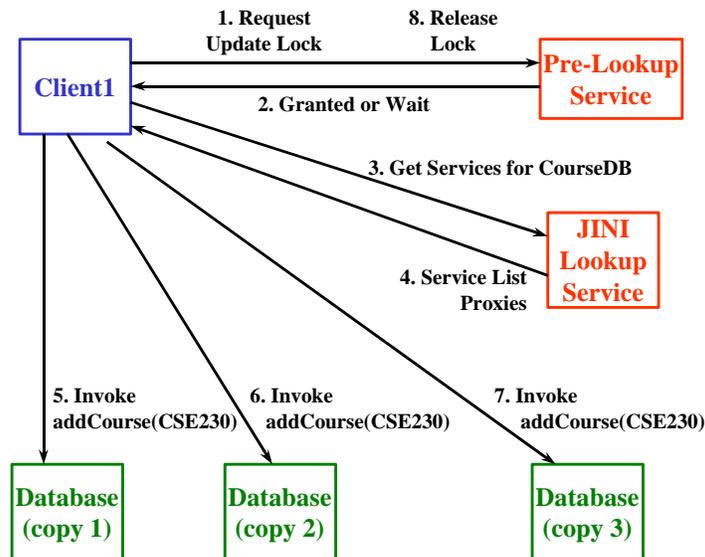


Figure 3: Execution Process in Prototype 6.

Prototype 6 incorporates a pre-lookup resource and associated services that implements a protocol that supports simultaneous reads in conjunction with at most one exclusive write, and includes PersonDB and CourseDB services for use by GUI clients. The pre-lookup services as shown in Figure 2, allow the locking and unlocking of services, identify clients (`getClientID`), and permit replicated database resources to register their services with the pre-lookup service (`addService` and `rmvService`). Thus, clients can still read the data even if one client is holding a write lock. PersonDB services are for authorization and authentication of the client, while CourseDB services allow course information to be queried and changed. Figure 3 illustrates the process and steps taken by a client. After startup, the client applications will be interested in discovering and utilizing services. In Prototype 6, prior to the JINI lookup service being consulted, the client must first interact with the pre-lookup service, as shown in Figure 3, arrow 1. The client consults with the pre-lookup service by discovering its existence and interacting with the JINI

## JINI: A Technology for 21<sup>st</sup> Century?

lookup service to obtain a proxy to request a lock. If a lock on the required service (read, insert, delete, or modify the CourseDB) is granted, the client can proceed according to arrows 3 through 7 in Figure 3. If a lock is not granted, the client is told to wait. The pre-lookup service will queue the client's identifier for the requested service to insure that starvation is prevented for clients that are denied locks at the pre-lookup service. Then, Client 1, in this case, enters a loop which will continuously request the lock (arrow 1) from the pre-lookup service. As long as another client holds the lock, a wait response will be sent to Client 1. Eventually the client holding the lock desired by client 1 will release the lock. When Client 1 next requests the lock and the first element of the queue for the service contains its identifier, Client 1 will be granted the lock, and processing proceeds via arrows 3 through 7.

### 4. Conclusions and Recommendations

Our conclusions and recommendations are constructed from a two-fold perspective. First, our efforts on the experimental prototypes have answered, in part, the questions posed in the introduction, specifically:

- **Can JINI Support Highly-Available Distributed Applications?** Yes, in fact Prototype 6 demonstrates that JINI can be utilized to architect solutions that are highly available.
- **Can JINI Support an Environment with Dynamic Clients and Replicated Services? Will Clients Continue to Operate Effectively if Replicated Services Fail?** Yes, in Prototype 6, it was possible to start and stop clients and stop and start resources. As long as JINI was given time to remove "failed" services, the clients and resources continued to interact effectively.
- **Can JINI be Utilized to Maintain "minutes-off" Data Consistency of Replicas?** Prototype 6 with the pre-lookup guaranteed that no updates would be lost if different clients attempted simultaneous updates.

The results are extremely relevant for present and future Army systems, and for distributed enterprise applications, in general, since the different architectural components of the prototypes can be cast as a new Java GUI, a legacy relational databases wrapped using JDBC/ODBC, and databases for authorization and general purpose information of interest to clients.

Second, is **JINI Ready for Prime Time?** That is clearly the question of interest. In our limited, yet concentrated evaluation of JINI, we have found many features that make it extremely attractive as a 21<sup>st</sup> century technology. Our reasons for believing JINI is ready for prime time include:

1. **Compatibility of JINI with Java write once run anywhere infrastructure.** The Java language and environment under which JINI operates is extremely homogenous, is operating system independent, and promotes interoperability between all of the components (clients and services) within the distributed application.
2. **Commitment of Sun to Java and JINI technologies, as evidenced by a recent keynote address by Chief Scientist Bill Joy [BJOY].** There is a significant commitment to JINI by Sun, and an expectation that JINI will play a major role in the Java arena in the coming years.
3. **Understandability and ease of use of JINI.** The individuals doing development had Java and database expertise, but no background in using JINI, Visual Café, and JDBC/ODBC. In 400 hours of work over the two month period of the work, six prototypes were designed and developed. This speaks to the ease of use of Java and JINI technologies.
4. **High-level abstraction nature of JINI API.** From a software engineering perspective, one of the major strengths of JINI is the ability to design a solution to a distributed application in terms of clients and the services that are required. This design can be constructed using a UML modeling tool. We believe that with JINI, UML modeling tools, and Java development environments, good software engineering practices and products can be attained.

However, our enthusiasm must also be tempered by the fact that our investigation, exploration, and evaluation of JINI is only in the initial stages. While our experiences have been mostly positive, there are a number of future work topics that must be explored in detail to arrive at a definitive conclusion.

- **Interoperability of JINI with critical technologies.** Will JINI work with legacy, COTS, and database assets? Will JINI inter-operate with CORBA and other distributed computing solutions? Can JINI and software agent paradigms successfully interact? All are critical to assess JINI's utility in 21<sup>st</sup> century.

## JINI: A Technology for 21<sup>st</sup> Century?

- **Verification of write-once-run-anywhere.** Is prototype of Section 3 extensible to Win95/98 and Solaris? Will Oracle, Informix, and other database platforms work? JINI's readiness for 21<sup>st</sup> century must be verified by conducting multi- and heterogeneous platform experiments.
- **Utility/robustness of other JINI technologies.** The list includes two-phase commit transactions, events in JINI, JINI's security model, and JavaSpaces, an API on top of JINI.
- **High-availability via multiple lookups and pre-lookup services.** Great care must be taken to explore, design, and implement prototypes that allow the incorporation of multiple lookup/pre-lookup services to have a reasonable and manageable impact on client applications.
- **Performance and scalability.** While our prototypes worked with 3 NTs, in practice, 10s, 100s, and even 1000s of clients and resources will need to interact. Consequently, the ability of JINI to scale and maintain performance in such a situation will be crucial.

Also, it is important to note that the JINI specification continues to evolve [JINISPEC]. Despite this cautionary note, based on our experiences and intuition, we believe that JINI has great promise and will be a successful and useful technology for the 21<sup>st</sup> century.

### References

[Arno99] K. Arnold, et al., *The JINI Specification*, Addison-Wesley, 1999.

[Edwa99] K. Edwards, *Core JINI*, Prentice-Hall, 1999.

[Free99] E. Freeman, et al., *JavaSpaces Principles, Patterns, and Practice*, Addison-Wesley, 1999.

[Morr97] M. Morrison, et al., *Java Unleashed*, second edition, Sams.net Publishing, 1997.

[Wald99] J. Waldo, "The JINI Architecture for Network-Centric Computing", *Communications of the ACM*, Vol. 42, No. 7, July 1999.

[BJOY] <http://www.javasoft.com/features/1999/07/bill.joy.html>

[JINI] <http://www.sun.com/jini/>

[JINIARCH] <http://www.sun.com/jini/whitepapers/architecture.html>

[JINISPEC] [http://www.sun.com/jini/specs/jini1\\_1spec.html](http://www.sun.com/jini/specs/jini1_1spec.html)

### Sample JINI Software:

<http://www.enete.com/download/> and <http://www.artima.com/javaseminars/modules/Jini/CodeExamples.html>

<http://www.jinivision.com> and <http://members.home.net/jeltema>

### JINI Tutorial:

<http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml>

### JINI-Related Information and Links:

<http://www.jini.org> and <http://www.eli.sdsu.edu/courses/spring99/cs696/notes/index.html>

<http://www.artima.com/objectsjini/introJini.html> and <http://www.artima.com/jini/resources/index.html>

### Link for JINI Installation:

<http://developer.java.sun.com/developer/products/jini/installation.html>