JINI: Evaluating the Technology and Impact on Present and Future Army Systems*

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

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

Abstract

Java has emerged as a critical technology for supporting the design and development of future applications that integrate and inter-operate with legacy, COTS, and database applications. The integration and interoperation require a network centric approach, since it is the network topology and infrastructure that must underlie any distributed computing application. Java RMI, remote method invocation, is an important component of Java that allows remote methods to be invoked on objects that are distributed across the network. JINI is a new architecture built on top of RMI that promotes the construction and deployment of distributed applications in a network centric setting. JINI provides concepts of leases, transactions, and distributed events to allow the construction of robust and scalable distributed applications. This white paper concentrates on the following issues:

1. What is the impact of JINI technology on present and future Army systems?
2. What is the JINI technology? Where does it fit into the overall Java picture?
3. Can JINI be utilized in a replicated environment which promotes high availability in a dynamic setting?
4. Is JINI ready for prime time? Will it be a big time player for Enterprise Computing Applications?

Multiple software prototypes have been developed to demonstrate and evaluate the operational capabilities of JINI running on a WinNT platform. Further, these prototypes are intended to address the questions posed in 3 and 4 above, thereby evaluating the current capabilities and future potential of JINI. The prototypes are an important first step in determining if JINI is “ready for prime time”.

* This report was supported as part of a research contract from the Mitre Corporation, Eatontown, NJ.
1. Introduction and Motivation

Distributed computing environments for the 21st century will require stakeholders (i.e., software architects, system designers, and application builders) to architect and construct solutions that facilitate the interoperation of new and existing applications in a network centric environment. 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 not simply to provide a means to allow different systems to interact and exchange information, but rather 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. Present and future Army Systems exhibit all of these characteristics and more, when issues related to the 21st century digitized force are examined and addressed.

The distributed computing platform has common characteristics with other domains, including the need to: manage and control data, allow humans to interact with the data, manipulate or modify the data, maintain the required levels of system availability and performance, and control the evolution of the computing enterprise. Currently, the data resides in a database or is under the control of a legacy/COTS application, with the interaction achieved via a forms-oriented interface, and the data manipulated by programs written in any number of programming languages. Interoperability in a client/server distributed computing environment includes the use of a multiple-language, object-oriented applications for distributed multi-process, heterogeneous platforms, which will allow information to be utilized in new and innovative ways.

The emergence of distributed object computing (DOC) technology and its competing and/or complementary paradigms such as DCE[OSF94, Rose92], CORBA[OMG95, Vino97, Yang96], and DCOM/OLE[Micr95], has opened new horizons to the object-oriented approach by enabling the parallel and/or distributed processing of large and computation-intensive object-oriented applications on heterogeneous hardware and software environments. This is precisely one of the major market segments that Java application development is targeting. Furthermore, DOC is positioning itself as the middleware that will enable a promising synergism between the object-oriented paradigm and innovative WWW technologies, which are two key facets for which Java has been championed.

Specifically, Sun supports many varied capabilities for interoperability as part of the Java Enterprise API [JAVAENT], including: Enterprise Java Beans, Java IDL for integration with CORBA, JDBC for Database Connectivity, JNDI (Java Naming and Directory Interface) for platform independent access to native naming and directory services, and Java Message Service for reliable queuing, publish/subscribe, and push-pull, etc. These different APIs take a traditional approach to interoperability by providing techniques and tools that support the interoperability of COTS, legacy, and databases, using standards, wrappers, middleware, and file services as a means to cobble together a “system of systems” thereby forming a distributed application. However, it is clear that today’s distributed applications are network centric, operating in a dynamic environment where clients, servers, and the network itself all have the potential to change drastically over time. 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. Sun has provided a critical first step towards this end, through its recently released JINI technology [Arno99, JINI, JINIARCH].

As a technology, JINI abstracts away from Java Remote Method Invocation (RMI) to provide a collection of functionality to support clients interacting with resources that offer services over a dynamically changing network. In a traditional client/server computing model, servers provide services via an API (set
of function calls) and export these services (export the API) for use by clients, which in turn must import
them. This can be accomplished in Java using the public methods in APIs and RMI. 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:

- Services can come (register and join) and go (leave) without impunity, since all interaction with
  services occurs via the lookup service.
- 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.
- The location of clients and/or services on the network can change at any time without impacting
  the network or the users.

The last observation is especially critical for present and future Army systems, which must be mobile and
react to the changing conditions of the 21st century battlefield. Further, it is important that there is a
divergence from business as usual. Distributed systems have been constructed with C programming using
remote procedure calls, with little or no reuse, yielding not extensible solutions. For future Army
systems, cobbling together solutions without method or reason is unacceptable and doomed to failure.

While questions that were posed in the abstract were conceptual in nature, the concrete goals and
objectives of our effort presented in the white paper can be posed as follows:

- Can JINI Support Highly-Available Distributed Applications?
- Can Replicated Database Services be Registered and Available for Use by Clients?
- Can JINI Support a Network-Centric Environment with Dynamic Clients and Services?
- Will Clients Continue to Operate Effectively if Replicated Databases 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?
- Is JINI Ready for Prime Time???

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?
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 premise of our approach is that multiple clients will be simultaneously interacting with multiple replicated database resources. Specifically, our prototypes include up to three replicated database resources interacting with one, two, or three clients via a single shared lookup service. We are concentrating on high-availability through replication, which may lead to situations where different clients see data that is “minutes” off in time, as a result of the replicas and clients modifying different replicas. However, it is critical that all replicated databases will eventually contain the exact same information, and that no updates are lost in the process.

The remainder of this white paper is organized into three sections and a conclusion to answer these questions. Section 2 provides background material on Java, including the concepts of object serialization and remote method invocation, which are two critical technologies that underlie JINI. Section 3 provides background material on JINI. This sets the stage for a detailed discussion in Section 4 of the six (6) prototypes that have been designed and developed, accompanied by reflections on our achievements and a comprehensive examination of potential future work. Section 5 concludes the white paper with recommendations on the readiness of JINI.

2. Background

This section gives a brief overview and conceptualization of Java accompanied by the critical topics that are needed to support the remainder of the white paper. An overview of Java is given in Section 2.1. Sections 2.2 and 2.3, respectively, review the object serialization and remote method invocation (RMI) capabilities of Java, two underlying Java technologies that are critical for JINI.

2.1 Overview of Java

Java is a third generation, general-purpose, platform-independent, concurrent, class-based, object-oriented language and associated environment, designed to be simple enough so that software engineers can quickly achieve fluency in the language and effectively utilize the language and environment for developing applets and applications. Java can be utilized in two distinct ways. In its original conceptualization, Java can be used to write special programs called applets that can be downloaded from the Internet and displayed/manipulated safely within a WWW browser. In addition, Java can also be used to design and develop standalone applications, with a wide range of capabilities and functionality. Platform independence is achieved in Java by using bytecode, which is similar in concept to the p-code used in early Pascal compilers that revolutionized compiler design and development. Thus, the Java compiler does not produce native executable code for a particular machine. Instead, the Java compiler produces a special format called Java bytecode that is finally interpreted to the native language of the host machine on the fly.

Java has two main components, the Java Development Kit (JDK) and the Java Runtime Environment (JRE). The JDK component is a package of programs and support files which is needed to develop Java programs. Included in the support files are class source code and documentation for the complete Java class hierarchy. The JDK contains the command-line driven javac Java compiler. The Java Debugger (JDB) is included with the JDK. It is also command-line driven, and has syntax similar to that of the UNIX dbx and gdb debuggers. The JRE component is needed in order to execute Java applications. It consists of the bytecode interpreter and other files such as the code verifier. JDK2 SDK v1.2.2 are available from Sun for Microsoft Windows 95/98/NT and SPARC Solaris platforms. In addition, there are a wide variety of platforms available as third-party ports[JDKP], including: AIX, Amiga, BeOS, Digital Unix, FreeBSD, HP 3000, HP-UX, IRIX, Linux, MacOS, Netware 4.1/IntranetWare, OpenServer 5, OS/2 Warp, OS/390, OS/400, Psion Series 3/EPOC 16, RiscOS, RiscBSD, and UnixWare.
In a normal compiled executable program, the object file contains the processor instructions to be executed, and the processor executes the instructions. In order to support platform independence, Java must provide an execution environment that can oversee the execution of applets and applications. The Java Virtual Machine (JVM) is utilized for this purpose. JVM is a program which runs on a particular hardware/OS platform (or ‘real’ machine) which interprets and executes a Java applet/application that is contained in a .class file. The .class file contains both executable JVM instructions (called bytecodes), and additional information such as the class structure, method and data member visibility, and superclass information. Since each JVM interprets the same set of bytecodes, true program portability is achieved by implementing JVMs for a wide variety of platforms.

There have recently been dramatic advances in Java compilers. Since the bytecodes are so well defined and so low-level, it is possible to compile sequences of bytecodes directly into native machine instructions. These just-in-time (JIT) compilers can supply dramatic performance improvements at runtime. Normal JVMs, when executing a class (via a .class file), create a table (known as a virtual table, or V-table) of method names pointing to method bytecode lists. When a method is executed, the bytecode list is looked up in the V-table and executed. With a JIT compiler, the V-table is still created, but each method name points to the JIT compiler itself, with the method's bytecode list in memory in a different location. When a method is executed, the JIT compiler compiles the required bytecode list to native processor instructions and then changes the link in the table so that the method name now points to the native code. The native code is then executed on successive method calls.

Java, through its public interface capabilities and package concepts also requires a clear definition of the exported portion of all classes/packages, which requires software engineers to specifically enumerate which packages, classes, and/or methods are imported. Thus, Java provides a set of application programming interface (API) packages. The Java 2 Core API, available online [JAVA2], is shown below:

```
java.applet           java.io           java.rmi.dgc      java.security.interfaces
java.awt              java.lang         java.rmi.registry java.sql
java.awt.datatransfer java.lang.reflect java.rmi.server   java.text
java.awt.event        java.math         java.security     java.util
java.awt.image        java.net          java.security.acl java.util.zip
java.beans            java.rmi
```

Each API contains a complete description of the package, which includes the classes and public methods that can be imported and utilized when developing Java applications.

### 2.2 Object Serialization

Persistence is the saving and loading of objects from some persistent storage mechanism (e.g., a database or a raw disk file), and is achieved in Java through serialization. Object serialization [OSER] is the conversion of an object into a string or some other data format that facilitates its storage to disk or its transfers across a network. The conversion back to an object from a serialized format is known as deserialization. Since Java already supplies methods for the serialization of intrinsic types, it is left to the software engineer to develop methods to serialize and deserialize a custom object; these methods are called respectively write and read. In most cases, the serialization of an object will consist merely of calling the write methods on all of the object's data members. If one of the data members is itself a custom object, there must also be a write method for that object as well.
2.3 Remote Method Invocation

The remote method invocation (RMI) API of Java is utilized for method invocations between client and server processes, when both are written in Java. Through RMI, it is possible to invoke methods on objects that are actually running under the control of a remote JVM. RMI is critical for both DOC and JINI. This has been recognized by others, namely: “This RMI facility, along with the CORBA IDL compiler libraries, make Java a very attractive platform for client/server applications.”[Morr97, p. 986]. The use of either RMI or CORBA allows access of objects over a network. Objects and object methods on a remote host can be accessed as though they resided on the local host. While IDL provides client/server access using the CORBA protocol, which allows access to any other server which uses the CORBA protocol, RMI provides a very robust and transparent interface to remote objects, but it is specific only to servers written in Java.

The designers of Java intended to keep RMI simple to use. Calling a method on a remote machine has all the simplicity of a remote procedure call (RPC) in C++. In Java, a remote method call is syntactically the same as a local method call; the implementation of the object which determines whether the method call is local or remote. The client objects implement “stub” code, and the server objects implement “skeleton” code. By using stubs and skeletons, which are strictly defined, Java insures that RMI code remains easy to implement and platform independent[Srid97]. RMI uses object serialization (see Section 2.2 again) to convert objects to streams of characters before transmitting them over the network.

3. JINI

In the past two years, Java has emerged as a strong candidate for constructing interoperable, distributed applications. Java’s capabilities of object serialization and RMI (see Sections 2.2 and 2.3 again), when coupled with the enterprise API, are a strong core of technologies for stakeholders (i.e., software architects, system designers, and application builders) to design, implement, and test distributed applications. The newest technology to enter the marketplace further extends and expands Java’s capabilities through a sophisticated API named JINI [Arno99, JINI, JINIARCH]. Stakeholders can utilize JINI to construct a distributed application by federating groups of users (clients) and the resources that they require. For future Army systems, the “groups” of users are various Army personnel (in divisions, brigades, battalions) and their resources could include weather data, map overlays, logistics data, and so on.

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. In Army systems, services would be defined to allow personnel and programs to access logistics data and weather data, and to supply map overlays. The services are registered with a look-up service which is essentially a clearinghouse for resources to register services and clients to find services. The registration of services can occur in a number of different ways 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. For example, in an Army system, the logistics data may only be available at the start of a battle or during specific, preset time slices. The clients, resources (and their services), and lookup service can all occupy the same or different computing nodes (workstations) in a distributed application.
JINI: Evaluating the Technology

Operationally, for stakeholders, JINI provides: an infrastructure for federating these services in a distributed setting; a programming model (i.e., resources registering services, the lookup service, clients seeking services) to support the definition and usage of services in a distributed setting; and the ability of the services to be available to any and all members (both clients and resources and their services). The programming model also includes transactions that support a protocol for two-phase commit for state changes of JINI objects. Functionally, JINI offers the ability of users to share resources (and hence, services) over a network without knowing their location, leading to the easy access of resources anywhere on the network and allowing the location of the user to change. This is especially important for Army personnel that may be moving their location while fighting a battle, resulting in a scenario where they would leave the network, move, and rejoin the network, and insuring that resources and services are again available. For stakeholders, this should simplify the construction, maintenance, and evolution of network centric applications. In the remainder of this section, we review the core capabilities of JINI, providing enough background and detail to understand the material in Sections 4 and 5. The material in this section is not meant to serve as a comprehensive treatment of JINI, rather, the reader is referred elsewhere [Arno99, Edwa99, JINI, JINIARCH, Wald99].

3.1 What is JINI?

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, which would occur in a battlefield situation. While they are connected, clients are guaranteed that either the visible services are available or that failure can be trapped and handled. Spontaneity from a resources perspective (and the services they provide), means that when resources fail, the network can adapt and change context, to insure that redundant services, if available, are now accessible to clients. Spontaneity is a key concern for present and future Army systems, since the interactions and movement of the different users, and the dynamic nature of the battlefield will require stakeholders to design solutions that seamlessly react to changes. 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. As indicated in the introduction to this white paper, such a scenario supports a number of laudable goals, namely: network plug-and-play of clients and services; erasing the hardware/software distinction – everything is a service; enabling spontaneous network centric applications; and promoting an architecture where the services define the functionality. The anticipated end-result is a simple and easy to understand and use infrastructure and accompanying technology.

The overall architecture that places JINI in context within the general workstation/PC computing environment is shown in Figure 3.1. In the figure, JINI is situated above Java technologies, which is reasonable, since JINI is a sophisticated API that offers a wide range of capabilities. Specifically, the JINI architecture depends on JVM properties, including: homogeneity, a single type system, object serialization, code downloading, and safety and security. Collectively, these properties insure that objects can be located and moved around a network in a consistent and trusted fashion, which is critical for JINI. The Application (clients) and Service (resources that provide services) all interact with JINI to comprise a distributed application. Figure 3.2 illustrates the codependence of JINI capabilities on critical Java infrastructure, programming model, and services. Clearly, the Java infrastructure of the JVM, RMI, and security, respectively, are key to insure that JINI operates according to bytecode verification within the runtime environment (JVM), clients and services (including lookup and other JINI services) can effectively communicate (RMI), and that actions taken by clients, services, or JINI will not violate the Java Security model. Java’s programming model (API and Bean concepts) influences the construction of JINI as an API that provides leasing, transactions, and events. Finally, critical interoperability services of
Java, JNDI (Naming and Directory Interface), EJB (Enterprise Java Beans), JTS (Transaction Service), and JMS (Mapping Service) are needed to prototype distributed applications using JINI, to provide the required tools for wrapping and interaction of legacy, COTS, and database components.

### 3.2 How Does JINI Work?

A distributed application constructed under the JINI framework is designed around the utilization of one or more lookup services, as shown in Figure 3.3. The lookup service is the central aspect 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. Thus, the lookup service is the clearinghouse where resources can “advertise” their services for location and utilization by clients. 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 enter the network/distributed application environment, it is their responsibility to locate and register with all appropriate lookup services. 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. Finally, note that, in practice, there is no way to prohibit a client for interacting with a resource independent of the lookup service; instead we must rely on software engineering discipline and practice.
After discovery has occurred, the resources must register their services with the lookup service. The resources register services on a class-by-class basis. Each class is registered as a service that contains one or more public methods. 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 3.4. In JINI terms, the service object is registered as a proxy, which is easily passed to the client requesting the service. The proxy contains all of the information that is needed to invoke the service. Once the resources
have registered their services, clients can discover and request services. In the request for service, shown in Figure 3.4, the client asks for the services based on the Java programming type. For example, a client will ask for the service to register for a course of the CourseDB class based on the signature of the method (i.e., the name, parameters, and return type of the method), namely: `status register_for_course(int)`. The lookup service will return a service proxy to the client, which is also shown in Figure 3.4. This proxy allows the client to invoke any or all of the methods defined within the service. Recall that a service can contain multiple methods which are all defined on the same class. Using the proxy, the client invokes the needed method(s) as it would any other Java method; the proxy allows the invocation to automatically utilize RMI so that the call occurs remotely on the computing node of the resource with the result of the call returned to the client. The interaction between the client and the resource occur independent from the lookup service; the lookup service simply facilitates the matching of client to service.

The final remaining concept of JINI that impacts on the usage of services by classes is leasing. A resource registers a service by granting a lease to the lookup service. A lease is 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, 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. This is facilitated in JINI using a provided lease renewal manager. The leasing and renewal process is intended to keep the registry fresh, containing all active and working services. This is of particular importance in a distributed application where resources leave the network due to failure or other reasons.

Once a resource leaves, the services should no longer be available. Expiration of leases is one way to make sure that this occurs. If a resource doesn’t renew its services, the lookup service will update its registry to remove the services. 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. In our prototyping experiences, there has been a 5 minute cycle for this process. That is, if a resource fails, it takes 5 minutes before its services are removed from the registry. In either of these two cases, 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 and implement clients that are able to foresee this situation. Whenever a service is invoked that is no longer available, an exception is raised. The client must be designed to recognize, trap, and reconcile this exception.

4. Experimental Prototype

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 backup service that can be utilized to support the client. The major assumption is that the services that are provided are replicated in multiple workstations, and each workstation maintains a replicated database resource in support of its services. 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 (if all clients stop activities), the databases will synchronize and contain the same information. It is crucial that updates not be lost during the modification and synchronization processes. We believe that these assumptions are typical of present and future Army systems for a number of reasons. First, there is a great deal of movement of clients (Army personnel) and accompanying reconfiguring of networks. Second, at different times during the battle, Army personnel (clients) may request data on demand to assess an evolving situation. Third, after the movement and rejoining, it is important that clients and resources be brought up-to-date, clients on the current data and resources (if replicated) on any changes and new data.

This section reviews our prototyping efforts on utilizing JINI to experiment with replicated databases and multiple clients in a dynamic network centric architecture containing multiple workstations running WinNT. To support the prototyping effort, an easy-to-understand university application is utilized, where students can query course information, and faculty can query and modify the schedule by adding, deleting, and changing courses. Consistent with the JINI technology as described in Section 3, resources that provide a set of services are defined to allow interactions with databases that contain person and course information. Once registered with the lookup service, these services are available for use by clients and other replicated databases. To fully explore the capabilities of JINI, a series of six prototypes have been designed, implemented, and tested. In the remainder of this section, the prototyping experience is detailed, focusing on basic assumptions (Section 4.1) and the functionality, capabilities, results, and achievements of each prototype (Section 4.2). Section 4.3 finishes with a comprehensive list and discussion of important future research.

4.1 Assumptions and Scenario

The experimental prototypes that have been developed are 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 provides the ability to query and modify them via JDBC. The resulting 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, as indicated in the introduction, there may be a time difference where the data in one CourseDB is minutes off the data in the other CourseDBs.

We believe that there are strong parallels between our experimental scenario and present and future Army systems. The GUI for student and faculty access is analogous to a new Java GUI that may be prototyped for a future Army application. The PersonDB for authorization has a parallel to an authorization database for a Army application, while the CourseDB represents a repository need by different types of users, which is similar to an Army database utilized by different personnel. Finally, PersonDB and CourseDB are relational databases (implemented in Microsoft Access) that are then wrapped using JDBC/ODBC so that they may easily interact with JINI. This would be analogous to an Army legacy relational database with a similar JDBC/ODBC wrapper being made available for use in a Java compatible application.

4.2 Prototypes

Our intent is to provide a series of prototypes that explore the capabilities and limits of JINI in support of present and future Army systems. The series of prototypes that have been implemented is as follows:

1. Initial prototype with one GUI client application and one database resource and its set of services for access to PersonDB and CourseDB via a Java server application. The client and database resource applications, and the JINI lookup service all run on the same NT workstation. The intent of this prototype was to gain familiarity with JINI, how it works, what it does, etc.

2. Second prototype that separates the client and database resource applications onto two NT workstations. The lookup service runs on the same workstation as the database resource. The intent of this prototype was to verify that the same JINI concepts function across multiple nodes.

3. Third prototype that extends the second prototype to three clients with a single database resource application running on three NT workstations. The lookup service runs on the same workstation as the database resource. The intent of this prototype was to verify that the same JINI concepts function across multiple nodes.

4. Fourth prototype that extends the second prototype to a single client (on one NT workstation) and three replicated database resource applications (on three NT workstations). The single lookup service will run on one of the NT workstations. The intent of this prototype was: to verify that a single client can interact with any of the replicated database resources; to determine if multiple replicated database resources can register the same set of services with the lookup service; to demonstrate that changes to one database resource (CourseDB) are then forwarded to all other replicas; and to experiment with failure by taking down one and then two replicas and demonstrating that the client can still access services of the remaining replica.

5. Fifth prototype that extends the fourth prototype with multiple clients (on three NT workstations) and three replicated database resource applications (on three NT workstations). The single lookup service will run on one of the server NT workstations. Failure support insures that when one or more replicas goes down, the system still works with the remaining replicas. The major difference from four to five involves consistency during database updates. Specifically, in the fifth prototype, a client receives the granting of all available services upon a lookup request, so that the client can then update all replicas at once. Thus, a client will modify all replicas rather than having the replicas modify one another.
6. Sixth prototype that extends the fifth prototype to explore a technique that eliminates consistency problems across databases by having a new pre-lookup resource (and services) that manages locking and exclusivity during updates. This insures that any client modifying one database will modify all databases, and those database will not be modified by other clients until the first client has finished. Thus, for example, there is no way for multiple clients to all register for the last seat in the same course.

In the remainder of this section, we examine the various prototypes, focusing on their purposes and goals, functionality, and the lessons that have been learned along the way. We have grouped related prototypes together to simplify and coalesce the presentation.

4.2.1 Prototypes 1, 2, and 3

The functionality of Prototype 1 included one GUI client (with limited capabilities) and one database resource and its services to allow usage of PersonDB and CourseDB via Microsoft Access. The client, database resource, and JINI lookup service all execute on the same NT workstation. The purpose of Prototype 1 was to allow the two individuals (graduate students) to learn and work with the new technologies (JINI 1.0 and Visual Café 3.0), to interact with Microsoft Access using Java and JDBC/ODBC, and to construct a baseline client/resource/services application for subsequent prototypes. The use of Prototype 1 was a good choice, since there were many intricacies in setting up JINI to work correctly on an NT workstation. Great care was necessary to set environment and classpath variables precisely, so that JINI and Visual Café would work correctly. All of this setup information was included as part of the readme file for Prototype 1 and subsequent prototypes. Since the client and resource (services and Microsoft Access) were all running on the same workstation, the passing of service to client occurred by sending code. The client did not need to use RMI but could simply execute the code in the local environment to interact with Microsoft Access. This approach was abandoned in subsequent prototypes, and we instead relied on the proxy approach as described in Section 3.2.

As an important aside, while Prototype 1 utilized downloading of code to allow the GUI to interact with Microsoft Access, in fact from a software engineering perspective it is not preferred, since it severely limits extensibility, introduces domain specificity, and could result in difficult to maintain code. In our specific situation, downloading of code requires that the database resource be located on the same node as the GUI. This is unrealistic in practice, since there will likely be tens and hundreds of GUI clients and only a few replicated database resources. In fact, when transitioning to Prototype 2, the services code for the database resources would no longer work, since the client and database resource were now on different computing nodes. In general, code for a service that executes on the client requires that all of the information that is needed is either available at the client, or that the service is complex enough to be able to interact with other resources on the network. The first case, can result in code that is difficult to design and change, since it must execute on a given computing node, as we found out when we evolved from Prototype 1 to 2. The second case, downloading code that interacts across the network, represents a poor design solution, since the service wasn’t a well-defined, distinct conceptual unit. If it is important to “move code”, a better solution would be software agents, which promote mobile code in dynamic environments. Mobile agents themselves may utilize the resources and services of JINI as clients. Finally, moving code is more costly, since the service that is moved may be quite “large” (number of bytes), and in limited bandwidth situations like Army systems, it is always better to make a “small” remote method invocation.

Prototypes 2 extended Prototype 1 by utilizing two NT workstations, one for the client and one for the lookup service and resource (and its database services). In addition, in this and subsequent prototypes, we utilized the proxy approach as described in Section 3.2 to register and deliver services. The only difference between Prototype 2 and 3 was the use of multiple clients in Prototype 3, as shown in Figure 4.1. The purpose of these two prototypes was to verify that JINI will work across the network and, in the case of Prototype 3, to insure that multiple clients can successfully interact with the same lookup service.
Since Prototype 3 subsumes 2 capabilities, we concentrate on its capabilities in the remainder of our discussion. As shown in Figure 4.1, the software architecture using JINI is very elegant and easy to understand. Prior to the architecture in Figure 4.1, there is a process that starts up JINI, the resource/database, and then the clients. Recall from Figures 3.3 and 3.4, that the resource, in this case the database, must discover and the join the lookup service to register all of its services, thereby creating the registry within the JINI lookup service. These services are listed in Figure 4.1, and are used to verify the user (student or faculty), query course information (students), and query and modify the course schedule (faculty). Clients will request services, receive service list proxies, and subsequently invoke methods (using RMI) as shown with the arrows labeled 1, 2, and 3, respectively.

Prototypes 2 and 3 were the first true network prototypes, since they involved multiple NT workstations. There were a number of achievements of Prototypes 2 and 3. From a system perspective, Visual Café was upgraded from 3.0 to 3.0c, which aided in the development of the remaining prototypes. From a client perspective, a full-fledged GUI was implemented that allowed students and faculty to access the CourseDB after authorization. The client application included queries to select, insert, update, and delete from the CourseDB, which is representative of most applications, including the military domains. From a client and resource perspective, broadcast discovery was utilized, to locate the lookup service which may be running on any machine(s) across the network. Finally, as previously mentioned, proxies were used to allow a client to easily connect and invoke services that involve Microsoft Access, which was executing on the workstation of the resource, which may be different from the client workstation.

4.2.2 Prototypes 4 and 5

The two missing aspects of Prototype 3 that are critical for the study were the lack of replicated database resources and the inability to experiment with the failure of replicas. Prototypes 4 and 5 were designed and developed to begin to handle these issues. In Prototype 4, the functionality included a single client and three replicated database resources, as shown in Figure 4.2. Each database resource runs on a dedicated, distinct NT workstation; the client and lookup service will also occupy different one of these same workstations. One purpose of Prototype 4 was to verify that a single client can interact with different database resources via the lookup service in a dynamic environment. Discovery of services by
the client may lead to the same or different resource in what appears to be a random fashion by the lookup service. Another purpose was to determine if multiple, replicated database resources can register the same set of services with the JINI lookup service. Redundant services are critical to allow for high-availability and dynamic behavior in a network centric environment. A third purpose was to demonstrate that changes to one database replica can automatically be forwarded to all other replicas, to maintain the consistency of the replicas. While the information in the replicas may be minutes off, over time, the intent is to insure that all database contain the same information. The final purpose was to experiment with failure, by taking down one and then two replicas and insuring that the client application can still discover, lookup, and invoke services regardless of the number of replicated resources. Figure 4.2 contains the services required to support Prototype 3. Note the set of services (addCourse2, removeCourse2, and updateCourse2) that have been added to Prototype 4 to support the interaction of the replicas to update one another.

![Figure 4.2: Services in Prototype 4.](image)

The execution process of Prototype 4 is shown in Figure 4.3. The first three steps to request services, receive a service list proxies, and invoke the addcourse() service, is the same as Prototype 3. However, the code of the addcourse() method has been altered for Prototype 4. Specifically, after the CourseDB has been updated on the replica, in this case copy 3, the addcourse() method then performs a number of other steps. First, it requests the CourseDB services needed to update replicas from the lookup service (labeled arrow 4) and is given the appropriate service list proxies (arrow 5) by the lookup service. Using these proxies, the addcourse() method invokes the addcourse2() methods (arrows 6a and 6b) for all of the other replicas. Upon the completion of the addcourse2() methods, the process finishes with a reply from the database resource (copy 3) to the client (not shown).

Prototype 5 diverges from 4 in its management of the replicas during the update process. In Prototype 5, the interactions of the database resources are eliminated. To insure that all three replicas are updated, the client now requests all database services from the lookup service. This request is answered by service list proxies for the “same” services on database resource copy 1, copy 2, and copy 3. Using this information, the client can initiate and manage the update process of the replicas, as shown in Figure 4.4. Notice that after steps 1 and 2, the remaining invocations initiated by the client update all of the replicas by adding the CSE230 course to the student’s schedule.
For both Prototypes 4 and 5, we conducted successful experiments with simulating failure by bringing down one and then two database replicas. In all of the experiments, the client was able to continue interacting with the remaining replicas. The one caveat involves the currency of the registry in the lookup service. Specifically, as mentioned in Section 3.2, regardless of the type of lease (forever or with expiration date), if failure occurred, it took a total of five clock minutes for the JINI lookup service to detect the failure and remove the affected services from its registry. To allow for the time period between failure of the resource and its removal from the registry, it is necessary to increase the complexity and logic of the client application software, so that a client is able to recognize and reconcile any raised exceptions. The effective utilization of leases and the ability of the client to react and adjust to dynamic changes using JINI should be the focus of future study, which will be discussed in Section 4.3.

The achievements of Prototypes 4 and 5 were significant. First, it was determined that it was possible to register the same set of services from database resource replicas with the lookup service. Since the replicas are running on different hosts, the lookup services can identify and distinguish among them. Second, it was found that a client can request and utilize all entries in a service registrar. For Prototype 4, the database resource replica can update all other replicas. For Prototype 5, this allowed a client to update all database replicas, eliminating the need for the resources to interact. Both Prototypes demonstrated that a client can continue to function in the event of failure of one or two replicas, as long as at least one of the replicas is alive. The client must wait for five clock minutes after failure to allow the lookup service to recognize the failure and adjust its registry. The consistency of the database is preserved in both prototypes through the update process of all replicas.
4.2.3 Prototype 6

Even though Prototypes 4 and 5 maintain consistency by updating all replicas, there is still the potential and strong likelihood for lost updates, if two or more clients are modifying the same information at the same time. This is a classic database problem that typically requires the utilization of database locking to ensure that updates occur in a serializable fashion. However, in a distributed, network centric application constructed using JINI, the replication of database resources significantly complicates this problem. Locking is neither simple nor straightforward, since the replicas are operating independently on different computing nodes. Prototype 6 is one possible solution to address the lost update problem for an application designed and constructed using JINI. Prototype 6, shown in Figure 4.5, incorporates a pre-lookup service that insures that only one client updates replicas at any given time. The pre-lookup service implements a protocol that supports simultaneous reads in conjunction with at most one exclusive write. Thus, clients can still read the data even if one client is holding a write lock. Clients must first interact with the pre-lookup service to secure a lock on any required services. Once the lock is obtained, the client will then interact with the JINI lookup service and replicas as discussed for Prototype 5. The purpose of this approach is to eliminate consistency problems and lost updates across replicas by requiring that locks be obtained in a centralized location (the pre-lookup service) prior to the discovery and invocation of services by clients.

Like any other resource, the pre-lookup service must discover, join, and register services. The pre-lookup services contains services as shown in Figure 4.5, to allow the locking and unlocking of services, to identify clients (getcClientID), and for the replicated database resources to register their services with the pre-lookup service (addService and rmvService). Unlike the earlier prototypes, the startup process has some differences in Prototype 6, which is illustrated in Figure 4.6. The JINI lookup service is started first, followed by the pre-lookup service which discovers and registers with JINI. Next, the database replicas can be started; Figure 4.6 illustrates three replicas discovering and registering. After this has occurred, one or more clients can start, and each client must also register with the pre-lookup service.
After startup, like in the other prototypes, 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 4.6, arrow 1. The client consults with the pre-lookup service by discovering its existence and interacting with the JINI 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 as with Prototype 5 (see Figure 4.4 again), and process according to arrows 3 through 7 in Figure 4.7. 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 makes its next request for the lock and the first element of the queue for the service contains its identifier, Client 1 will be granted the lock, and processing will proceed via arrows 3 through 7.

![Diagram showing the execution process in Prototype 6.]

The achievements of Prototype 6 mirror Prototypes 4 and 5, as presented at the end of Section 4.2.2. The major new achievement or Prototype 6 is the incorporation of the pre-lookup service and its locking scheme that guarantees that updates will not be lost since only one client is able to secure a lock and modify the database at any time. Other achievements include the potential versatility of the pre-lookup service to support other locking protocols, according to the requirements of the application domain. For example, in some situations, a write excludes reading protocol might be relevant. Care was taken to design the pre-lookup service so that all requests for services, read or write, must be locked before the client can request the services from the JINI lookup service. The main drawback of this approach is that there is no way to force the software engineers writing client code to obtain locks from the pre-lookup service prior to interacting with the JINI lookup service. Instead, the approach relies on software engineering discipline and practice to insure that clients are designed according to requirements. Another drawback of this and all of the prototypes include the absence of multiple JINI lookup and pre-lookup services, adding redundancy in the primitives needed for JINI to work effectively. A final drawback is a lack of support for restarting a failed database replica, which will likely require logging and recovery techniques. These and other topics are the subject of future work, which is discussed in Section 4.3.

4.3 Future Work

The work presented in Sections 4.1 and 4.2 is a prelude for future exploration into the capabilities and potential of JINI and related technologies. Hence, future work related to this effort includes following:
In Prototypes 4, 5, and 6, leasing was utilized to register services with the lookup services. We attempted a variety of leases, including FOREVER leases and ones with specific expiration dates. Leases with expiration dates must be renewed before they expire by the service. We also experienced interesting behavior of the lookup service upon failure of a database. Whenever a database failed when using FOREVER leases, it took five minutes of clock time for the lookup service to remove the service from its registry. During that five minutes, any client requests for failed services will result in an exception. It is important to note that there will always be situations where a resource has failed and the registry still contains the service, since there is a time lag between the failure and the lease expiration. Further, there are bound to be situations where a proxy is delivered to a client, and prior to the client executing the method, the resource fails. Clearly, the remedy will require the complexity of the client code to be increased so that it can adjust in the event of failures and exceptions. This is important in Army systems where it is necessary to limit the availability of sensitive data, and will require further investigation.

Java has been touted for its write-once-run-anywhere philosophy, and it is important to verify that this holds true using JINI. The dissemination of Prototype 6 to multiple operating system/hardware platforms should be undertaken. Movement of the clients to Win95, Win98, and Solaris platforms is possible. The lookup and pre-lookup services can also be moved to non-WinNT platforms. The current database services are restricted to Win98/98/NT platforms, since we have used Microsoft Access. However, our use of JDBC/ODBC should allow us to replace Access with another database platform (e.g., Oracle, Informix, etc.) to clearly demonstrate that JINI works across platforms and architectures. In fact, a reasonable Prototype 6 might use four clients (Win95, 98, NT, Solaris) and up to 6 database replicas (combinations of Win and Solaris Platforms as well as Oracle, Informix, and Access), with the platforms for the pre-lookup and lookup services being changed as well, to fully test the capabilities of JINI to support diverse architectural options. To be totally convinced of JINI’s readiness for Army systems, it will be important to conduct these multi- and heterogeneous platform experiments.

While we have utilized the concepts of discovery, join, registration, and leasing in the prototypes, there are many other JINI capabilities that should be investigated for their potential to support high-availability distributed applications. For example, it is possible to organize services according to clients, so that if there are multiple, replicated services, the client can be automatically routed to the “best” service (e.g., closest on the network) to meet their needs. JINI also supports a two-phase commit transaction model, to allow operations to be grouped so that success means all of the operations executed, while failure means none did. The use of transactions may be an alternative to the pre-lookup service, allowing a client to group update transactions for multiple database replicas in a transaction that either succeeds or fails. Experimentation is needed to insure that starvation of clients using transactions will not occur. Recall that for Prototype 6, the pre-lookup service maintains a queue of waiting client requests to insure that starvation is prevent. There is also an evolving JINI security model, through which it is possible to grant or deny access to resources by clients. In a battlefield situation, the granting and especially denial of access may be critical to insure that strategic information does not fall into enemy hands. The potential role of the JINI security model in support of resource protection and control is also an important future work topic.

The pre-lookup service (see Section 4.2.3) is currently designed with a concurrency control model that supports exclusive write while allowing multiple simultaneous reads. Future experimentation can explore other locking protocols to evaluate their impact on the overall structure and behavior of the system. Also, the granularity level of the locking in the pre-lookup service is very coarse. Current locks occur by service, and there is a single service defined for inserts, deletes, and modifications to CourseDB which requires that all of these methods be locked for the entire database, regardless of the actual course (e.g., CSE230). This is unrealistic.
in practice, and further experimentation is needed to explore the registration and locking of database services in a more fine-grained manner. Granularity is important for Army systems, since different personnel may see different information based on their overall placement in the command hierarchy.

- To truly support high availability, redundancy of lookup services and pre-lookup services as realized in Prototype 6 must be fully explored. According to JINI documentation, multiple lookup services can be supported. In our brief experimentation with this feature, we have found that the client quickly becomes confused when multiple responses to a lookup request are received. One obvious solution would be to increase the logic of the client code; but, this must be carefully balanced to insure that the client code remains simple and easy to design/develop. There is nothing that prevents one lookup service from registering with another lookup service, thereby supporting a nesting of lookup services. Another important issue concerns the pre-lookup service, which is instrumental to insuring database consistency in the replicated repositories. For high-availability, multiple pre-lookup services must be provided, which in turn will require pre-lookup services to likely communicate and exchange information as we transition to a distributed concurrency control protocol. Thus, it makes sense to experiment with multiple lookup services; the full three client/three server scenario with one, two, and even three redundant JINI lookup services. Nested lookup services and multiple pre-lookup services, and apropos combinations, must also be considered. The utilization of multiple lookup and pre-lookup services has the potential to increase the complexity of the client code, which is a detriment from a software engineering perspective. Great care must be taken to explore, design, and prototype techniques that allow the incorporation of multiple lookup/pre-lookup services to have a reasonable and manageable impact on client applications. Replication was one of our main goals and objectives for this study, as indicated in Section 1, and we have only explored replication of the database resource and not of JINI functions and capabilities.

- While the prototypes have demonstrated that the failure of a replicated database will automatically result in remaining database services handling future requests, the issue of restart for the failed database has not been addressed. Logging and recovery of database transactions is a well understood area of computing. In the context of the prototypes, the log would contain the function calls that have been made to modify (insert, update, or delete) the Course and Person databases. This log would need to include timestamps for each function call. New services would need to be added to each replicated database resource to allow it to retrieve the log from another replica. These services would be utilized by each “failed” replicated database upon restart after registration with the JINI lookup service. Once retrieved, the log would be used to update the database for all timestamps since the failure. There are many issues that must be considered in this process, most notably, the role of the pre-lookup service as supported in Prototype 6 must be understood. There is an important time period that the restarted database will require to retrieve and process the log. During that time, it is critical that all other updates to replicas also be made on the restarted database. Further, if multiple lookup services are supported as discussed in the prior bullet, then the impact of logging and restart must be examined. As Army personnel and resources enter and leave the system, logging is critical to insure that clients (personnel) receive current data and resources can be quickly brought up-to-date.

- JINI, as shown in Figure 3.1, is an API built on top of Java that provides functionality for leases, transactions, and distributed events to allow the construction of robust and scalable distributed applications. Using JINI as a core, it is possible to build other, more complex and sophisticated services to suit many diverse purposes. The JavaSpaces technology [Free99, JAVASP] is a powerful new service for JINI from Sun to promote the construction of distributed applications. JavaSpaces takes a shared memory approach and provides specific areas where persistent objects can be exchanged among various processes that are sharing data and coordinating their activities.
There are many diverse capabilities that are supported by JavaSpaces, including: distributed data structures, synchronization techniques, loosely coupled communication, message passing, channel data structures for communication, leases and automated lease renewals, and handling partial failures with distributed transactions. Investigation of JavaSpaces is a logical next step to explore the ability of the technology to support replicated databases and associated requirements of the various prototypes.

- In all of the prototypes, the individual performance and overall performance of the executing application was always quite deliberate. Given the state of the network at UConn (i.e., hubs, routers, etc.), it was unclear if the internal network structure or JINI/Java were the cause of the slow performance. Thus, another logical extension of the effort is performance analysis of the prototype systems. This analysis can start with Prototypes 5 and 6 and proceed through Prototypes that are developed to experiment with different locking protocols, multiple lookup services, logging, recovery, and restart, and the JavaSpaces API.

This list of potential future research and development efforts related to JINI and JavaSpaces outlines an ambitious yet reasonable approach to further investigation and evaluation of the ability of JINI/JavaSpaces to satisfy the needs and requirements for the interoperability of and construction of present and future Army distributed applications.

5. Conclusions and Recommendations

Our conclusions and recommendations are constructed from a two-fold perspective. First, our efforts on the 6 experimental prototypes presented in the white paper have answered to a great degree the questions posed in the introduction, specifically:

- Can JINI Support Highly-Available Distributed Applications? Yes, in fact Prototypes 4, 5, and 6 all demonstrate that JINI can be utilized to architect solutions that are highly available.
- Can Replicated Database Services be Registered and Available for Use by Clients? Yes, this was demonstrated starting with Prototype 4, which registered multiple database resources used by a single client, and in Prototypes 5 and 6 which contained multiple clients.
- Can JINI Support a Network-Centric Environment with Dynamic Clients and Services? Will Clients Continue to Operate Effectively if Replicated Databases Services Fail? Yes, in all of the prototypes starting with 4, it was possible to start and stop clients and stop and start database resources. As long as the JINI registry was given time to remove “old” or “failed” services, the clients and resources continued to interact effectively.
- Can JINI be Utilized to Maintain “minutes-off” Data Consistency of Replicas? Prototypes 4, 5, and 6 all demonstrated this to varying degrees. Prototype 6 with the pre-lookup was superior, since it also guaranteed that no updates would be lost if different clients attempted simultaneous updates.

The results are extremely relevant for present and future Army systems since the prototyping architecture as presented in Section 4 has strong parallels with such systems, i.e., 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. Finally, it is important to again reiterate the significant scope of future work reviewed in Section 4.3, which clearly outlines the limits of our study.

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 for
dynamic network centric applications, which typify the needs and requirements of present and future Army distributed applications. Our reasons for believing JINI is ready for prime time include:

1. Compatibility of JINI with Java write once run anywhere infrastructure. The homogeneity of JINI with Java is a significant plus. Other DOC solutions like CORBA require the purchase of an ORB from a vendor, which must be installed and maintained on all appropriate nodes. While IDL provides the basis for a common middleware, the interaction between the different nodes is decidedly heterogeneous. In the various prototypes that have been implemented, we have demonstrated the ability to incorporate a GUI client, which is representative of a new Java application, which interacts with a database resource and its services, which is representative of a legacy database application. That is, the usage of Microsoft Access for the Course and Person databases is analogous to interacting with legacy databases. Services are defined, which, via JDBC and ODBC, allow a Java interface to be established to the relational data stored in Access. In the case of the prototypes, any database (e.g., Oracle, Sybase, etc.) could be substituted with anticipated similar results. 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. The approach using JINI requires all legacy and COTS to be wrapped using Java. In a CORBA-complaint solution, there would be multiple programming languages and multiple operating system platforms; a heterogeneous environment that is more difficult to maintain and evolve. This is a significant advantage of Java and JINI over competing DOC technologies. Note that while Sun has promoted the write-once run anywhere of Java/JINI, as we note in future work, it is important to conduct comprehensive verification experiments.

2. Commitment of Sun to Java and JINI technologies, as evidenced by a recent keynote address by Chief Scientist Bill Joy [BJOY]. According to Joy, there are four key technologies, two of which are directly relevant to this white paper: “The first of these is the vision of having objects everywhere, which has been realized with Java technology. The second is to be able to have systems with these objects connected together into spontaneous networks. We're realizing that vision with JINI technology.” This is further elaborated by: “The idea of JINI technology very simply is to use one of the most powerful of the ideas from distributed systems: the idea of agents. Each device or service is an object represented by an agent which moves in the network. These agents are not defined by protocols, they're defined as programming language types. So I don't have to you to implement a protocol, I can simply pass you an executable agent over the network.” Clearly, 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. Track record of Java and Sun. Clearly, over the past two years, Java has exploded onto the computing landscape, and has enjoyed many successes in areas which include: electronic banking, electronic commerce, information dissemination (push and pull), agent computing paradigms and environments, Java Beans and EJB, and so on. This proven track record in conjunction with Sun’s commitment to the entire Java family of technologies, is a strong predictor of the potential success and utility of JINI.

4. Understandability and ease of use of JINI. The involved individuals doing development had Java and database expertise, but no background in using JINI, Visual Café, and JDBC/ODBC. Over the two month period of the work, these two graduate students made substantial progress, each putting in 200 hours in support of the design and development of the six prototypes presented in Section 4. This speaks to the ease of use of Java and JINI technologies, which is an important concern. In prior experiences with other students on other CORBA related projects, we have observed much longer time lines to acquire and apply knowledge in a productive manner.
5. High-level abstraction nature of JINI API. From a software engineering perspective, the high-level nature of the JINI API is conducive to constructing well-designed clients and services that will inter-operate as a distributed application. Remember, it is always possible to create a solution that will satisfy the operational and functional requirements of a specification. Most distributed applications are built in a bottom up fashion, using distributed application builders. Software engineering is definitely lacking in distributed application development, despite the arrival of UML modeling tools (e.g., Rational Rose [ROSE], Rose Real-time [ROSE], Paradigm Plus [PARA], ObjecTime Developer [OTIME], etc.) that allow distributed and real-time requirements to be partially specified. 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 conducted using a UML modeling tool. Once designed, the transition to an implementation can be accomplished via a development environment, such as Visual Café, which was used for the prototypes presented herein. The key is to promote good software engineering practices, which lead to well-designed and easier to maintain products and applications. 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. Clearly, the future and potential work outlined in Section 4.3 should raise the caution flag for the immediate adoption of JINI for present and future Army distributed applications. While our experiences have been mostly positive, it will be important to see of over the coming months the future work and issues raised in Section 4.3 can be explored in greater detail to arrive at a definitive conclusion. Also, it is important to note that the JINI specification continues to evolve [JINISPEC], and changes have the potential to impact on performance and usage. Java’s biggest drawback over its lifetime has been the incompatibility of new versions of the APIs with prior versions of the APIs (re. deprecated APIs), which has caused significant difficulty in maintaining Java applications to work with the newest releases of Java. Perhaps the release of Java 2 will stabilize the language and APIs. 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 21st century.
References


Additional JINI References

Sample JINI Software:
http://www.enete.com/download/
http://www.jinivision.com
http://members.home.net/jeltema

Java and JINI Tutorials:
http://java.sun.com/docs/books/tutorial/

JINI-Related Information and Links:
http://www.jini.org
http://www.eli.sdsu.edu/courses/spring99/cs696/notes/index.html
http://www.artima.com/objectsjni/introJini.html
http://www.artima.com/jini/resources/index.html

Link for JINI Installation:

JINI Technology 1.0 API Documentation:

Principles of Distributed Computing: