Jini Technology Applied to Railway Systems

Txomin Nieva [a, b, *], Andreas Fabri [b], Abdenbi Benammour [a]

[a] Institute for computer Communications and Applications (ICA)
Communication Systems Dept. (DSC)
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne, Switzerland
http://icawww.epfl.ch
{txomin.nieva, abdenbi.benammour}@epfl.ch

[b] Industrial Software Systems CHCRC.C2
Information Technologies Dept.
ABB Corporate Research Ltd.
CH-5405 Baden, Switzerland
http://www.abb.ch/chcrc
{txomin.nieva, andreas.fabri}@ch.abb.com

Abstract

In the world of pervasive computing where large management systems, as well as small devices, all become interconnected, the problem of the configuration and management of networks is becoming increasingly complex. System administrators have to deal with many problems due to the increasingly distributed architecture of systems. Jini, a new paradigm for the development and management of distributed systems, provides mechanisms that enable systems to plug together to form an impromptu community. This practice report demonstrates how Jini can be applied in an industrial environment, or more exactly how it can be used in the integration of embedded devices on-board trains in the back-office IT infrastructure of railway operators. We present two use cases: the first is about an on-board service that allows for remote access to an on-board diagnosis database; the second is about automatically installing new services on-board trains. The results are encouraging and prove that Jini is the appropriate technology to link application servers and service gateways in embedded servers.

1. Introduction

The Internet, having caused a revolutionary impact on office automation, is currently provoking a large effect on industrial automation and information systems. The emergence of the Internet provides a framework that enables for communication with any piece of hardware and/or software, regardless of where it is physically located. Heterogeneous distributed embedded systems, which were commonly isolated in the past, are increasingly connected to networks and integrated within information systems. The management of distributed embedded systems has become a huge task for embedded systems providers, operators and service organizations that want to offer to their customers a high quality of service. The interconnection of distributed embedded systems and information systems brings significant benefits and offers new business opportunities.

The problem is that the configuration and management of networks is becoming increasingly complex. System administrators have to deal with many problems due to network unavailability, systems down, high latency and other typical problems of any distributed system.

Recently, *Sun Microsystems* proposed a new paradigm for the development and management of distributed systems. This new technology, called Jini, provides simple mechanisms that enable systems to plug together to form an impromptu community - a community put together without any planning, installation, or human intervention. Each system provides services that other systems in the community may use. Jini simplifies interactions with networks, by eliminating the cost of manual administration of services.

In the framework of an R&D project [1] for the railway manufacturer *Adtranz*, we experimented with Jini technology in order to evaluate it in an industrial application. We identified two potential application domains. The first application domain concerns the embedded devices on a train. They are interconnected with a train communication network. As the composition of a train changes over time, the communication network must configure itself automatically when the train is powered up, and it should reconfigure automatically when locomotives and cars are added to or removed from the train.

[*] Corresponding author.

The second application domain is the integration of the on-board systems of a vehicle fleet into the back office system of an operator and a manufacturer. Typical examples are off-board databases that store disturbance data for all devices of all vehicles of a fleet for the entire life cycle of the fleet. These systems generate work orders and allow for the development of new maintenance strategies based on data analysis. Another typical example is the remote management of on-board services. Installation, removal, or updates of on-board services is very complex and expensive because operators have to schedule train stops to allow maintenance staff to go on-board a train and perform the tasks manually.

The back office integration application domain also has a dynamic behavior. Sometimes, this communication provides a high latency and low bandwidth, e.g., when the train is only reachable via GSM. Other times, the communication is only possible at certain times, e.g., when a train is in a train station or a garage equipped with a WLAN. Furthermore, the services available on each vehicle may vary among the vehicles in the fleet.

In this project we concentrated on the back office integration. We present the main results of a research project done by the *ICA* institute of *EPFL* and *ABB Corporate Research* in close collaboration with the railway manufacturer *Adtranz*. The objective of this project was to study the applicability of Jini in two use cases of railway systems. The first is an on-board service that allows for the remote access to an on-board diagnosis database with diagnosis information. The second is the automatic installation of services on-board trains.

We implemented these use cases on a demonstration platform in the laboratory. As we used the same equipment as the equipment used in the field, a future deployment to the field should not be a problem. By using the same equipment as the equipment used in the field we ensure that the results obtained from our experiments are reliable. However, a future implementation in the field would provide additional information, which cannot be obtained in the laboratory, in regards to the behavior of the system in a real environment.

This paper is organized as follows. First, we briefly present the main concepts of Jini. Second, we develop in detail the two use cases. Then, we discuss some relevant aspects from the analysis and design of these use cases. Finally, we present some conclusions from the actual work.

2. Jini

Jini technology [2-4] provides a simple infrastructure for providing services in a network. It enables spontaneous interactions between applications. The result is a network of services connected together dynamically. Services can join or leave the network in a robust manner. Clients can rely upon the availability of visible services. The purpose of Jini is to federate groups of hardware and/or software components into a single, dynamic, distributed system. The resulting federation provides the simplicity of access and ease of administration. It guarantees the reliability and scalability of the whole system. Let us first describe the goals of Jini:

- **Network plug-and-play**. A service is visible after it is plugged into the network. There is no need to configure the system. The "network" announces the availability of a new service. Any interested client is then able to use the service. To deploy a new service, we have only to plug it into the Jini-enabled network.
- **Spontaneous networking**. When services plug into the network they become available spontaneously. They can be discovered and used by clients and by other services. Clients and services work in a flexible network. They can organize themselves in the most appropriate way for the set of services that are actually available in the environment. When a plugged service is disconnected, the network withdraws automatically the service.
- **Service-based architecture**. Products can be designed as services instead of stand-alone applications. As Jini enables services to collaborate with each other to perform particular tasks, service developers gain in reusability and modularity.
- **Simplicity**. Jini is concerned with how services connect to one another. There is no constraint on what services provide and how they should work. Jini is based on Java. However, services can be written in languages other than Java, if they provide a chunk of Java code that can participate in the Jini mechanisms.
- **Reliability**. Jini supports interactions between distributed services. It helps programs to find services, and ensures spontaneous availability. Services can appear to and disappear from the Jini federation in a very lightweight way. Interested parties can be automatically notified when the set of cooperating services changes. When damage occurs in the network, Jini is able to repair itself.

We now briefly present the key concepts of Jini. The main concepts of Jini and their relationships are described in the conceptual model shown in Figure 1.

- **Jini Service**. A Jini service is an entity that can be used by a person, a program or another service. A service may be a computing program, a hardware device or a component of the Jini system. Members of a Jini system federate to share access to services. A Jini system consists of services that can be collected together for the performance of a particular task. The dynamic nature of a Jini system enables services to be added or withdrawn from the federation at any time. Services in a Jini system communicate with each other by using a service protocol represented by a set of interfaces (written in Java).

- **Service Item**. A Jini service has a service item. A service item is an object that represents this service in the Jini federation. A service item is composed of a service identifier, a set of attributes and a proxy object.
- **Service ID**. A service identifier is a global unique identifier for a service. This identifier is assigned by a lookup service (further detailed) the first time a service registers in the Jini federation. Once a service ID is assigned to a service, the service must remember it.
- **Service Proxy**. A service proxy is an object that encapsulates the mechanisms that a service and a client communicate with. When a client system looks for a service it receives from a lookup service a proxy object enabling the communication with the requested service.
- **Service Attribute**. Service attributes represent relevant characteristics of a service; relevant features that distinguish one service from another in ways that are not reflected by the type of the proxy. In this way, Jini clients can perform rather complex searches of services based on these attributes.
- **Jini Group**. Jini services are structured within groups. A group usually represents a rather small (typically the size of a workgroup) community of services. The default group is called "public". Group names are only unique within the naming space of a network.
- **Jini Federation**. The Jini federation is an abstract concept that represents the full set of communities (or groups) of Jini Services.
- **Jini Client**. A Jini client is a system that uses a Jini service. A Jini client can eventually be another Jini service or just another entity. In an ideal Jini federation there would be nothing but Jini services that collaborate to perform certain tasks.
- **Lookup Service**. Services are found and resolved by a *Jini Lookup Service (JLS)*. The JLS is the central bootstrapping mechanism for the system. It provides the major point of contact between services. It is essentially a process that keeps track of all of the services that have joined the Jini community. The JLS resembles a *name server*. However, the matching is more powerful. We can search for services that implement some interfaces, or belong to special classes. We can also specify a search based on service attributes. Services register themselves within a JLS by publishing their *service item.* Jini can be seen as an implementation of the *Broker* architectural pattern described in [5] by *Buschmann et al*., where the JLS plays the role of the broker.

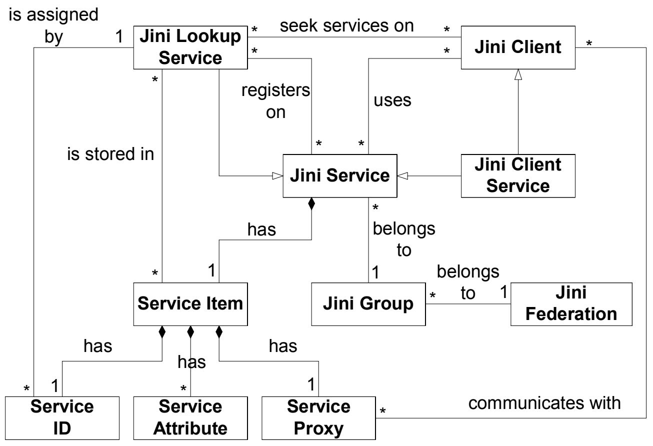

Figure 1. Jini conceptual model

Finally, we describe the key mechanisms of Jini.

- **Discovery Protocol**. Entities that wish to start participating in a distributed Jini system must first obtain references to one or more JLSs. The protocols that govern the acquisition of these references are known as the discovery protocols. A Jini discovery protocol is the means by which Jini entities find Jini communities. There are several discovery protocols that can be used according to each particular situation. The "*Multicast Request Protocol*" is employed by entities that seek to discover nearby (on the local network) JLSs. The "*Unicast Discovery Protocol*" is used when an entity already knows the particular JLS it wishes to talk to. The "*Multicast Announcement Protocol*" is used by JLSs to announce their presence. As a result of the discovery process, a Jini entity disposes of references to JLSs.
- **Join Protocol**. A Jini entity, using references to JLSs obtained during the discovery process, can advertise the services it offers. The *join protocol* regulates how services join Jini communities. A service joins a JLS by publishing its service item. The first time a given service joins a JLS, the JLS will assign it a *serviceID*. The service must remember this and use it when it registers itself with all JLSs in the future.

- **Lookup Protocol**. A Jini entity, using a reference to a JLS obtained during the discovery process, can search all the service items provided by this JLS to find services of interest. This search can be based on the type of the service proxy, on the unique identifier of a service, or the attributes contained in a service item.
- **Leasing**. In distributed systems, there are situations when different parts of a cooperating group are unable to communicate - either because one of the members has crashed or because the connection between the members of the group has failed. To deal with these problems, the notion of *lease* was introduced. Rather than granting services or resources until that grant has explicitly cancelled, a leased resource or service grant is *time based*. When the time for the lease has expired, the service ends or the resource is freed. The time period for the lease is negotiated when the lease is first granted. Leases may be renewed or cancelled before they expire by the holder of the lease. In the case of no action, caused by a service crash or a network failure, the lease simply expires.
- **Remote Events**. Jini entities occasionally need to be notified when certain interesting changes happen in their environment. To do that, Jini has the notion of remote events. An event is an object that contains information about some external state change that an entity may be interested in, e.g., an expected service joins the Jini community. The notification is sent to listeners that have subscribed interest in receiving a particular event.
- **Transactions**. Jini supports the notion of transactions. Transactions provide a way to group a series of related operations so either all the operations succeed, or all the operations fail. Jini uses a two-phase commit process. First, a transaction manager signals each of the participants to go into a *pre-commit* phase. Then, each of these participants notifies the transaction manager whether the operation was successfully completed or if it failed. If any of the operations failed, the transaction manager tells every participant to *abort*. Otherwise the transaction manager tells the participants to *commit*, which causes them to make their changes permanent. Jini only defines the transaction protocol as an interface but it does not implement the protocol. The transaction protocol must be implemented by each service.

3. System Architecture

In this section, we introduce the system architecture that we used to implement our Jini-enabled use cases. This corresponds to the architecture we used successfully in several projects over the last years ([6, 7]), before introducing Jini technologies. This system architecture, shown in Figure 2, is composed of three-tiers:

(i) An on-board train *service gateway* is connected by one way to the train communication network, e.g., TCN [8], and by other way to the Internet using a wireless network, e.g., via GSM or WLAN. This gateway machine runs on-board services. As an example, a service provides remote access to the on-board diagnosis database.

(ii) A *ground station* acts as a middle tier between clients and gateways. This ground station is an application server that offers stand-alone services to users, but also services that make use of remote on-board train services. In the latter case, the application server consolidates and dispatches requests to associated trains. The ground station is also a router that automatically establishes communication links with gateway machines, either via GSM or WLAN.

(iii) Finally, a *client machine*, using nothing but an Internet browser, uses services offered by a ground station; independently these services are stand-alone services or services that make use of remote on-board train services.

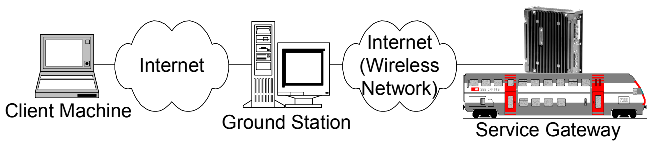

Figure 2. System architecture before applying Jini

The administrator of the system is responsible for updating, removing or adding services on the gateway and ground station machines. Due to the distributed nature of the system, this is a complex and tedious task, especially when a service must be installed on hundreds of trains. Another problem is that as trains are mobile systems, remote on-board services may be available only during certain periods of time, e.g., when a train enters a train station with a WLAN. It is difficult for a ground station to manage this dynamic behavior of services. In addition, the clients' perception of the quality of service of the application server is poor because ground stations may offer services that are not currently available.

In order to be able to run Jini enabled services, it is necessary to introduce at least one machine, running a Jini Lookup Service (JLS), connected in the same LAN to the ground station. Eventually, we could also run the JLS on the same machine as the ground station. Additionally, some other JLSs all over the Internet, train stations and garages may be used. These JLSs would forward service subscriptions and other events to each other by tunneling. Tunneling is just an optimization mechanism, which for simplicity it is not described in this paper. The new system architecture is shown in Figure 3.

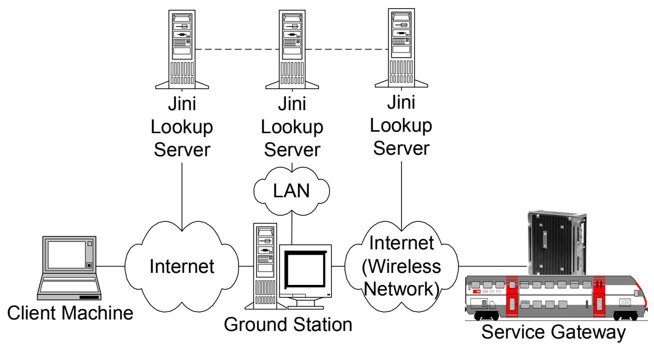

Figure 3. Jini-enabled system architecture

4. Use Case 1: Access to On-board Diagnosis Database

This section presents the use case to provide remote access to an on-board diagnosis database. First, we describe the system that existed before we applied the Jini technology. Then, we discuss problems we encountered with this system. Finally, we describe the Jini enabled system.

4.1. System Description

The use case corresponding to the system that existed before we applied the Jini technology is shown in Figure 4.

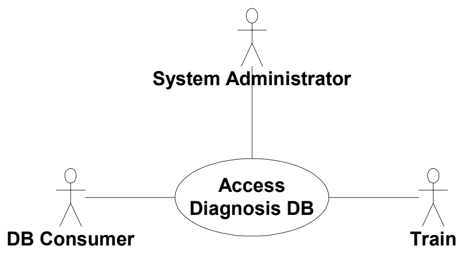

Figure 4. Access to on-board diagnosis DB use case before applying Jini

The actors that play in this use case are:

- DB Consumer: an entity (typically operator maintenance staff) that wants to get access to diagnosis information stored on the on-board database.
- Train: a train with an on-board diagnosis database that collects diagnosis data.
- System Administrator: responsible for manually setting up the system. For example, the system administrator has to find out which are the currently available trains and how they can be accessed.

An on-board service allows the access of the on-board diagnosis database to a service in a ground station. This system is shown in Figure 5.

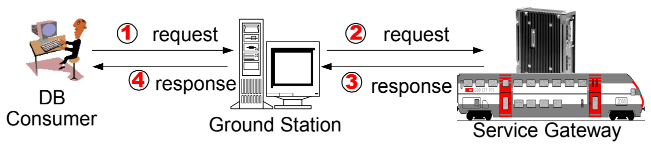

Figure 5. Access to on-board diagnosis DB system before applying Jini

The sequence of actions is the following:

1. A DB Consumer contacts a ground station and requests access to a database on-board a particular train.
2. The ground station consolidates and dispatches the requests to the corresponding train gateway.
3. The train gateway processes the request and sends the response back to the client.
4. The ground station consolidates and sends the response back to the client.

4.2. Problems

We encountered the following problems with this system:

- Trains are not always available when requested: thus, the services that access on-board diagnosis databases are not always available either. Moreover, these services are not able to automatically notify the system when they are available or not.
- High configuration work done manually by the administrator: the administrator must manually configure associations between train identifiers and their current location on the network.
- Static and non-flexible architecture: the architecture does not respond well to dynamic changes on the configuration. As all connections are established

manually, if a connection is lost there is no automatic reconfiguration. In addition, there is no way to enhance the accessibility of a train service, e.g., when a train enters a rail station with a WLAN.

4.3. Description of the Jini-enabled system

The use case corresponding to the Jini-enabled system is shown in Figure 6.

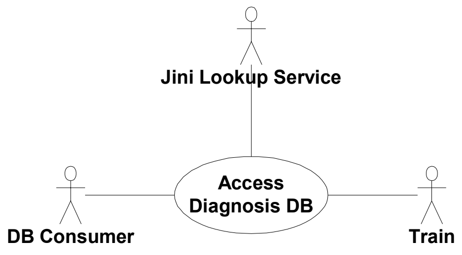

Figure 6. Jini-enabled access to on-board diagnosis DB use case

In this use case, the JLS replaces the system administrator. The JLS allows us to automate the actions that were manually performed by a system administrator. The JLS allows on-board services to automatically register within the Jini community. On-board services are lease based and they must renew periodically their lease in order not to be removed from the community. A ground station acts as a Jini client. It subscribes to receive notifications when on-board services appear or disappear from the community. The new Jini-enabled system is shown in Figure 7.

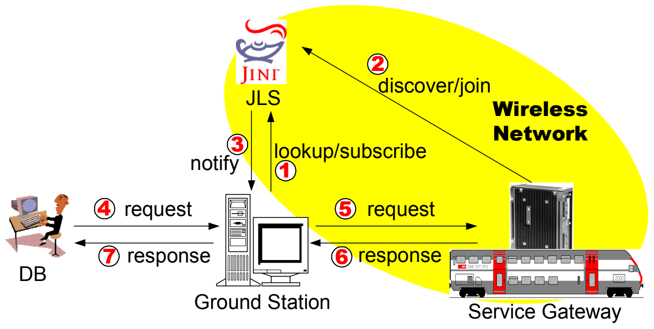

Figure 7. Jini-enabled access to on-board diagnosis DB system

The new sequence of actions is the following:

1. A ground station registers in the JLS for a particular remote event, namely an on-board DB access service joining the federation. This registration enables the ground station to be notified when trains take part in the Jini community.
2. After startup, a Jini service on-board a train discovers and joins the JLS.
3. The ground station receives a notification in form of a remote event that contains a reference to the proxy object of the service.
4. Clients contact the ground station and request access to a database on-board a particular train, which figures in the set of registered trains. This set of trains is up-to-date as trains that do not renew the lease of their joint operation are removed from the system.
5. The ground station searches for the corresponding proxy, which in turn contacts the on-board service on the associated train.
6. The on-board service processes the request and sends the response back to the proxy in the ground station.
7. The ground station sends the response back to the client.

5. Use Case 2: Automatic Deployment of Services

This section presents the use case to deploy new services on-board a train. First, we describe the system that existed before we applied the Jini technology. Then, we discuss the problems encountered with this system. Finally, we describe the Jini-enabled system.

5.1. System Description

The use case corresponding to the system that existed before we applied the Jini technology is shown in Figure 8.

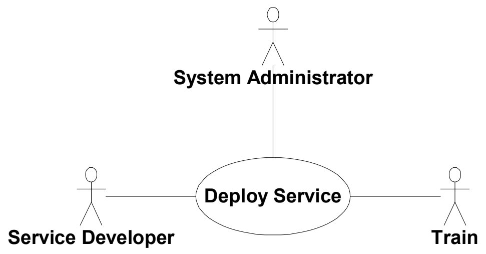

Figure 8. Deploy service use case before applying Jini

The actors that play in this use case are:

- Service Developer: a service developer that wants to deploy a service to a train.
- Train: a train where a service must be deployed.
- System Administrator: this actor is responsible for manually setting up the system. For example, the system administrator has to find out which are the currently available trains and how they can be accessed.

A service developer submits a service to a ground station. Then, the ground station installs the service on the corresponding train. This system is shown in Figure 9.

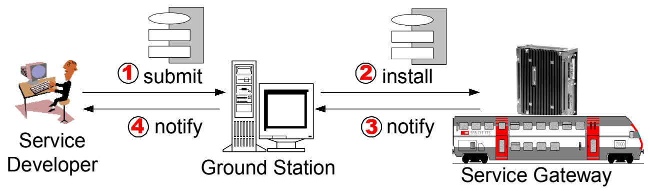

Figure 9. Service deployment system before applying Jini

The sequence of actions is the following:

1. A service developer submits a service to a ground station.
2. The ground station installs the service on the corresponding train.
3. The train notifies to the ground station that the service has successfully been installed on the train.
4. The ground station sends the notification back to the service developer.

Note that the described system is an oversimplification of a real system. A real system should keep records of versions of services already installed on vehicles. Furthermore, a real system should address security issues, e.g., to manage people authorized to install services or to establish times when services may be installed, started, stopped, or uninstalled.

5.2. Problems

The encountered problems are basically the same as described in the first use case, mainly due to unavailability of trains at certain times. In addition, this use case reflects the problem that in the case a service developer wants to install a service on a fleet of trains, he has to perform the process manually for each train, which is rather inefficient and error prone.

5.3. Description of the Jini-enabled system

As in the first use case, the JLS is a new player in the use case. The JLS replaces the system administrator. This fulfils the requirements of automatic configuration of the system. Additionally, we introduce a *Temporary Service Store*, where services are temporarily stored until they have been installed on all target trains. The new use case is shown in Figure 10.

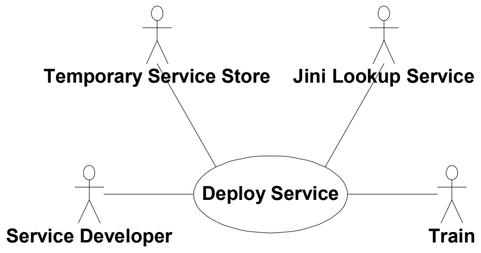

Figure 10. Jini-enabled deploy service use case

A JLS, which represents the Jini federation, allows train installation services to automatically register within the Jini community. In addition, the Temporary Service Store stores a service locally until it is installed on all target trains. The new Jini-enabled system is shown in Figure 11. For simplicity, the Temporary Service Store is in the same machine as the ground station, which it is not always necessarily the case.

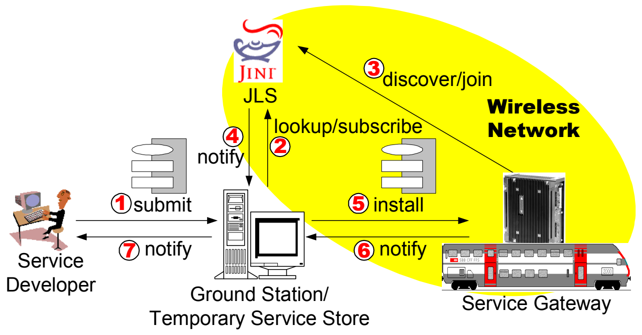

Figure 11. Jini-enabled service deployment system

The new sequence of actions is the following:

1. A service developer submits a service (typically as a bundle or package) to a ground station. He specifies the identifier of each target train.

2. The Temporary Service Store registers with the JLS to be notified when trains with installation services join the Jini community.
3. When a train enters a rail station, its installation service initiates a *discover and join* process with the JLS.
4. The join is notified to the ground station. The notification is stamped with a proxy object of the installation service.
5. The Temporary Service Store gets the service installation proxy object and contacts the installation service on the target train to install the service on the train.
6. The installation service notifies the Temporary Service Store that the installation has been successfully done.
7. When all target trains are updated, the Temporary Service Store notifies the service developer.

6. Discussion

We implemented both use cases in a demonstration platform in the laboratory. However, in order to have reliable results, we used the same equipment as the equipment used in the field. For our experiments, we used the version 1.0.1 of the Jini implementation of *Sun Microsystems*. The on-board gateway is based on *Sun Microsystems*'s Java Embedded Server [9], a service gateway that runs on any Java platform. Service gateways allow system administrators to remotely manage (add, remove or update) on-board services. Currently, service gateways are under standardization by the Open Service Gateway Initiative (OSGI) [10]. Jini does not replace service gateway functionality. Rather, Jini services can enhance service gateways by enabling automatic management of services. A good example of this is offered by the automatic installation service use case. In this use case, we enhance the service gateway by a Jini-enabled installation service. This service is installed on the service gateway. When this service becomes active (typically when the service gateway is powered on), it registers in the Jini community enabling the automatic deployment of pending services to these service gateways from a "Temporary Service Store". Deployed services can be either simple services or Jini-enabled services. During this project we demonstrated that Jini and service gateways are complementary technologies. This is in concordance with what the OSGI [10] says:

"The OSGi specification works with various device access standards and is compatible with and can enhance a JINI environment... An OSGi compliant system provides an excellent focal point for the deployment and management of JINI services."

The first use case provides an efficient multi-user remote monitoring service. This service allows authorized clients to browse data from any registered train with location transparency. As the communication protocol is embedded within the proxy object of a Jini service, the communication protocol used to communicate with this service is transparent for a client entity. This gives service developers a great flexibility because they can implement services based only in services' API ignoring the actual communication protocol. Thus, Jini offers an ideal framework for implementing distributed applications that use heterogeneous communication protocols.

The only currently available implementation of Jini, provided by *Sun Microsystems*, is based on Java Remote Method Invocation (RMI) [11] middleware. This may dissuade the use of Jini in certain cases, e.g., when there is a firewall between services and their clients. This is not a problem in our application as the ground stations, as well as the embedded servers, are in the same domain.

We take full advantage of Java's dynamic class loading feature. The classes of the service proxies are dynamically loaded from the embedded servers. This allows for the different implementations of service proxies on different vehicles, with only different versions of the same class. Not only the class of the service proxy can be downloaded, but also classes of objects referenced by the proxy. The proxy can hence be arbitrarily complex and interact with the application server that hosts it. When classes are loaded dynamically, security becomes an issue. This is another reasons why Jini has not taken off yet. It is not so much an issue in our system as we only use our own services.

The current implementation of Jini needs a Java2 Runtime Environment. This dissuades the use of Jini on small devices with few megabytes of memory available. In fact, this was one of the reasons we did not investigate the first application domain mentioned in the introduction: Controllers in devices hooked on the train communication network, typically, are too small to host a Java virtual machine. Lack of memory and CPU was not an issue in the back office integration application domain, the embedded server is an industrial PC.

Another reason we did not Jini enable the controllers, was the fact that the train communication network does not support TCP/IP. We then had to first develop the Jini protocol for this particular network. This could be an interesting approach to investigate but it was not in the scope of our experiments.

Finally, the experiments demonstrated that the current implementation of Jini does not perform exceptionally well. The JLS provided by *Sun Microsystems* is slow to register new services and notify service registrations to service listeners. Jini services and clients implemented with the current libraries provided by *Sun Microsystems* have a too long response time. However, it is reasonable to expect that future implementations of the JLS and Jini libraries by *Sun Microsystems* and/or third companies will speed up response times improving considerably the performance of

Jini services. Therefore, we consider the performance as a minor transitory problem due to the early versions of the current implementations of Jini.

7. Conclusions & Future Work

The use of Jini technology simplifies the development of distributed systems because Jini forces distributed systems developers to deal with the network in early stages of development. Jini is not just a programming library to implement distributed systems, but a new paradigm for distributed system development. Using Jini, distributed systems developers can automate the, usually tedious, configuration process of such systems. Jini enables the search for particular services based on complex attributes. Jini provides self-healing communities of services as it uses the concept of leases. Regardless of minor problems of the current implementation of Jini and the lack of standardization of services, we demonstrated that Jini offers an efficient approach for developing distributed applications. We also demonstrated that Jini technology and service gateways are complementary technologies.

Jini is a young technology. Even if the concepts are well defined, currently there are few Jini services available. Jini will have real success when service developers can dispose of a huge number of standardized services all over the world. Before Jini becomes ubiquitous, it will be found in distributed applications as the one described in this paper: where the application is a service provider and, at the same time, the only user of these services. As our experience with Jini was positive, we recommend the use of Jini for implementing services in distributed industrial applications.

We plan to deploy our experiments in the field in order to validate them in a real environment. This deployment should not be a problem as we used the same equipment as the equipment used in the field. We believe that the experiments in the field will validate the results obtained in the laboratory. Experiments in the field will provide additional information, which cannot be obtained in the laboratory, in regards to the behavior of the system in a real environment. Although there are no major technical problems that impede the implementation of Jini in a real environment, there are some issues that must be addressed before deploying Jini in a real environment. These problems mainly concern administration tasks (such as how to identify train equipment and how to define appropriate attributes that enable a particular train, vehicle or equipment to be found) and some security issues (such as defining who is allowed to seek a particular train equipment).

Acknowledgements

Thanks to *Adtranz* who brought a real framework to the discussions and the implementation of our hypotheses, and to *Holly Cogliati*, from *ICA*, for proofreading this paper.

References

[1] A.Benammour, Swiss Federal Institute of Technology (EPFL), "*Jini-enabled Trains (JET)*", Diploma Project, March, 2000, http://icawww.epfl.ch/nieva/DiplomaProjects/99-00/Abdou/jet.htm.

[2] K.Arnold, B.O'Sullivan, R.W.Scheifler, and J.Waldo, "*The Jini specification*", Addison-Wesley, 1999.

[3] K.Edwards, "*Core Jini*", Prentice Hall, 1999.

[4] Jini User Group, "*Jini Homepage*", 2000, http://www.jini.org.

[5] F.Buschmann, R.Meunier, H.Rohnert, P.Sommerlad, and M.Stal, "*Pattern - Oriented Software Architecture: A System of Patterns*", Wiley, 1996.

[6] R.Itschner, C.Pommerell, and M.Rutishauser, "*GLASS: Remote Monitoring of Embedded Systems in Power Engineering*" in IEEE Internet Computing, vol 2, 1998.

[7] A.Fabri, T.Nieva, and P.Umiliacchi, "*Use of the Internet for Remote Train Monitoring and Control: the ROSIN Project*" presented at Rail Technology '99, London, UK, September 7-8, 1999, http://icawww.epfl.ch/nieva/thesis/Conferences/RailTech99/article/RailTech99.PDF.

[8] IEC, "*Electric Railway Equipment - Train Bus - Part 1: Train Communication Network*", IEC 61375-1, 1999.

[9] Sun Microsystems, "*The Java Embedded Server*", 2000, http://www.sun.com/software/embeddedserver/.

[10] OSGI, "*Open Service Gateway Initiative Homepage*", 2000, http://www.osgi.org.

[11] Sun Microsystems, "*Java Remote Method Invocation White Paper*", 1999, http://java.sun.com/marketing/collateral/javarmi.html.