
Ford Motor Company

PsiNaptic Inc.
216, 200 Rivercrest Drive SE
Calgary, Alberta
Canada T2C 2X5
T: 403 720-2531
F: 403 720-2537

www.psinaptic.com

## The Paper Offer: `04ANNUAL-64`

## Title: Integration of In-Vehicle Components Leveraging Jini Technology

## Author Info:

**Vladimir Rasin, Ford Motor Company**

**Steven Knudsen, PsiNaptic Inc.**

## Abstract

Vehicle manufacturers face the increasingly complex task of integrating and interfacing various components in the vehicle, both factory-installed and aftermarket, while the interfaces to such components and their interdependencies are becoming more complicated. The complexity is aggravated by the necessity to maintain a variety of vehicle models and platforms. Perhaps the greatest challenge is associated with the task of integrating consumer devices in the vehicle, since vehicle manufacturers do not have control over the functionality available in such devices, yet consumers will request this functionality to be available (and integrated) in the vehicle. Therefore, vehicle manufacturers must respond to current technology demands and anticipate those in the future.

In this paper we describe a solution developed by Ford Research to manage the interaction of in-vehicle modules and devices in a standardized (across Ford Motor Company) manner. One of the important building blocks of this solution is based on Jini™ network technology. Jini, a service discovery protocol, enables transparent, dynamic integration of modules and devices by providing an infrastructure that hides the complexity associated with such integration. Jini technology by itself is not new and has been under evaluation by Ford Research for its merits for in-vehicle use for some time. We will describe the main issues and conclusions of our evaluation process, including our initial reservations about the feasibility of incorporating this technology into real vehicles.

Ford Research has partnered with PsiNaptic to implement in-vehicle Jini-based technology. Their novel Jini Lookup Service implementations significantly extend the boundary of where and how Jini technology can be used in the vehicle. We will describe how this technology can be used not only in the infotainment space but also in the more traditional automotive domain. We will discuss how in-vehicle components, including sensors and actuators, can be integrated in a cost-efficient manner and in such a way as to create an easily controllable, dynamically configurable in-vehicle electrical architecture.

**Consumer Devices In the Vehicle**

The automotive development cycle is much longer than that for consumer technologies, and it will remain this way even as vehicle manufacturers try to shorten the vehicle development cycle and decrease the time between initial vehicle design and vehicle delivery. It is therefore safe to assume that OEMs will always have to deal with the issue how to integrate consumer devices in the vehicle.

There are a few main driving forces beyond this convergence. First, there is the increasing customer demand to be able to use devices and modules, already possessed by the customer, in the vehicle. It is possible, of course, to supply a vehicle with embedded phone, but if the customer already has a phone he would like to leverage the devices and infrastructure he already has. The in-vehicle phone will never completely replace the regular phone. Therefore, seamless integration of consumer devices in the vehicle will increase customer satisfaction.

Another important aspect of such integration is the role that OEMs want to play in such development relative to the aftermarket industry. Integration of consumer devices in the vehicle opens the field for all new in-vehicle applications. To be able to offer and control such applications will require OEMs to come up with a way to make consumer devices usable in the vehicle. What more complicates the task in this case is that we do not know a priori what kind of services mobile devices will bring to the vehicle. For example, the customer's phone may provide GPS functionality that in- vehicle applications could use. The OEM solution, therefore, must be flexible enough to accommodate consumer devices with a variety of available services and functionality.

The seemingly easiest way to use consumer devices in the vehicle is to install a 'cradle' inside the vehicle where the user can install his or her device. The problem with this solution that it will work only with certain device models/brands. It's very difficult to make a prediction as to what kind of device a customer is most likely to use in the car in today's environment, and practically impossible to make such assumptions for a few years down the road.

This is an example of a traditional, 'local computing' approach (as it applies to the in-vehicle environment) towards this problem. The vehicle is supposed to be self-sufficient, which is very difficult, if not impossible, to achieve in this context. Another way of looking into this problem is to think of the vehicle as part of a potentially sophisticated networking environment in which mobile devices are just other nodes on the network. We do not make any assumptions about the nature of the network, particular protocols, etc. We do allow the notion that some vehicle nodes, in this case mobile devices, may at a later time bring some additional functionality to the vehicle, thus enriching the services available to the customer

There are many examples of successful solutions and deployments, mostly in the enterprise space, that deal with such distributed infrastructure. These solutions differ in the way they handle the fact that they are no longer operating in a pure local computing environment. Some architectures hide the remoteness from the applications. Applications do not know that they are using services on a remote device/computer. This is the way CORBA and DCOM operate. This approach -- "think local, work distributed" -- has a certain issues associated with it

- Network latency. Treating a remote method call as a local call could introduce errors,
- Concurrency. Normally requires additional software, which could be a problem in the resource-constrained in-vehicle environment.
- Address Space. Working with remote objects as they are local may introduce memory-related errors and corrupt memory.
- Partial Failure. In local computing, one component's failure is usually a total failure. In a distributed environment, one component's failure is a partial failure. Such failure should not lead to system/network failure.
- Tight coupling between client and service provider. Applications on the client have to know in advance who is the service provider and its ties to the specific service implementation.

To really be able to integrate mobile devices with the vehicle environment, the solution must provide for a reliable, dynamic and deterministic way of interacting with remote network nodes.

**Jini Overview**

[1]

Several technologies such as Jini™ , Remote Method Invocation (RMI™), Universal Plug and Play (UPnP) [1][2][3][4], peer-to-peer technologies like JXTA™ [5], SMS, and others, purport to support the distribution and management of software-defined services and information in network environments. Each has its strengths and weaknesses, which are well documented elsewhere [6].

Jini network technology addresses the problems of distributed software-based services and information (as described in [2]

part by the so-called seven fallacies of network computing ). It tackles these problems by defining mechanisms to support the federation of machines or programs into a single, dynamic distributed system. Devices participating in such a system can enter and leave at will, can tolerate network and system variability, and can offer "services" and resources to other devices and systems in the federation. In the context of this specification, a "service" refers to an entity that can be used by a person, group of people, organization, program or other service. The service can be anything that can be offered by a computational, networked device, including access to a network, computation, storage, information, access to hardware (such as a printer, modem, etc.) or another user.

A Jini federation consists of one or more devices, one (or more) of which hosts a Lookup Service (LUS). The LUS is used to keep track of services made available by devices in the federation and is used by devices to discover and obtain services.

*What happens when a client device enters a Jini federation?*

1. The Jini client device is connected to a network on which a device is running a LUS.
2. The client “discovers” the LUS via the usual Jini mechanisms (multicast or unicast discovery).

[3]

3. The client receives the ServiceRegistrar Java object and loads it into its JVM.
4. The client uses the ServiceRegistrar object to find and obtain services of interest, register its own services with the LUS, or request notification of changes in the federation.

The main steps taken by a service provider and a service consumer in a Jini network are shown in Figures 1, 2, and 3. Each client device (service provider and service consumer) uses the Jini Discovery mechanism to obtain a ServiceRegistrar object. That object is used to either register a service or search for and upload existing services. Once the service is activated by the consumer, it and the provider may communicate directly via the service. A more thorough overview of the

mechanisms in and operation of the federation can be found in [2].

**Figure 1** The initial interaction between a service provider and a LUS. The order of operations is provided by the numbered text.

**Figure 2** The service consumer discovers the LUS, obtains the ServiceRegistrar software object, and uses it to find and upload a desired service (or services).

**Figure 3** Once the service consumer has activated a service, the consumer and the provider may communicate directly (via the service).

Jini networking technology applications cover areas as diverse as military communication systems, network element management for cellular and land telephony, medical device monitoring, add-on service provisioning at gas stations, industrial equipment control, and many others.

## 1.1 Jini in the vehicle

The list of in-vehicle applications that could be enabled by Jini technology is very broad. The typical examples are usually centered on consumer devices, such as phones or pdas. In this case, the vehicle has a LUS in which the consumer devices register their services. The offered services are not necessarily matched one-to-one to devices. The same device can provide a few different services. For example, a phone could contain such services as 'contacts', 'calendar', 'gps' and/or just 'phone'. Each of these services will be registered with the LUS of the vehicle or, potentially, even with the LUS located somewhere else, e.g. on the server. The clients of these services could be a telematics unit (using the 'phone' service to dial a service provider), a navigation system, hands free phone application, etc. At the same time the phone itself could be a client of other services available in the vehicle and/or provided by the vehicle.

Another simple, practical example of a useful Jini application is remote diagnostics. For example, a vehicle may have a diagnostics information service and be running its own LUS. In this case, the vehicle (self-) registers its service with the LUS, thereby making it available to Jini client devices. One such client may be the user’s local dealership. The dealership would be able to obtain the diagnostic service wherever and whenever the vehicle has access to a network. The network could be a wide-area network (via a cell phone, for example), a wireless LAN at the user’s home, or a network in the dealership shop. Because the service is a self-contained software object that hides its implementation details, the dealership would not have to worry about having the right software installed to be able to talk to the vehicle—the diagnostics service takes care of such details. Other potential clients of the service include the user (say, via a laptop or PDA at work or home), other dealerships (when the user is away from home), all the way up to the vehicle manufacturer itself.

Ford Research & Advanced Engineering was very interested in Jini technology because of its ability to facilitate the provisioning of software-based services for and from the in-vehicle hosting platform (telematics unit, radio, etc.). Moreover, Jini seemed to offer a solution to the “late-binding” problem. Systems that offered services based on prescribed software interfaces could be introduced into designs well into the product cycle. Since services are really just software objects, implementation details and even hardware details could be abstracted and hidden from dependent systems so long as they adhered to the interfaces. Moreover, the implementation could change over time without affecting other systems.

Jini was initially touted as a technology for all devices on a network, and in particular for dedicated computing platforms such as printers, storage media, cameras, and such. However, the reference implementation by Sun Microsystems, based on

RMI, was simply too large for resource-constrained devices. This was especially true for automotive platforms. Modern telematics units, with the most computing resources in an automobile, cannot afford the several megabytes of storage required by the reference implementation.

This was confirmed by the vehicle prototype developed by Ford Research Laboratory, which was presented at the Convergence 2000 auto show. This prototype included Jini-based implementation over an IDB-C in-vehicle data bus. Although the technology showed great promise in general, it was deemed unrealistic at that time to use it in production for the reasons mentioned above.

**Jini for Resource-constrained Devices**

In 2001, PsiNaptic Inc. completed a Jini-compliant implementation of the Lookup Service (named JMatos™) that solved the footprint problem. JMatos allows very resource-constrained devices, such as telematics platforms, to host Jini services. Instead of three or more megabytes, the JMatos LUS requires less than 100 kilobytes of storage. This compares favorably to the tens of kilobytes required by typical services.

JMatos is designed to off-load as much of the processing from the LUS host and service offering devices as possible. This fundamental architectural shift led to the implementation of the ServiceRegistrar as a self-contained object, rather than a proxy. That is, the ServiceRegistrar is an object requiring little subsequent collaboration with the LUS. This is in contrast to RMI-based ServiceRegistrar implementations, where the majority of the processing is performed by the LUS.

By implementing the ServiceRegistrar as an object, the LUS functionality has been isolated and reduced to the following functions:

- Performing the discovery process.

  a) Supporting the multicast discovery process. This involves sending periodic multicast announcement messages and receiving multicast request messages.

  b) Supporting the unicast discovery process. This involves constructing and marshalling a ServiceRegistrar object over a unicast stream. While a JVM with full serialization capabilities simplifies this process, less capable implementations can rely on internal serialization logic that produces a byte stream compatible with Sun’s serialization specification.
- Maintaining a distributed services cache. The LUS must,

  a) Provide a time-based (leased) storage for services. Expiration of the lease results in the removal of the cache item. A persistent storage mechanism is used to restore the cache between system starts. This guarantees the lease semantics of the Jini specification.

  b) Refrain from service content manipulation. All serialization/marshalling of the service occurs within the JVM of the registering agent. The LUS simply stores the serialized ‘bytes’ of the service.

  c) Provide a synchronization mechanism when a change to the LUS cache occurs.

The ServiceRegistrar object (offered by the LUS) performs all remaining Jini functionality:

- Maintains a copy of the services cache. The cache is synchronized with the ‘master’ copy (maintained by the LUS)

when appropriate.

- Performs all lookup activities (i.e. query for a service item matching a particular pattern) using the services cache copy.
- Controls the join protocol – that is, serializes the service item and transmits the resulting bytes to the 'master' services cache (on LUS) for storage.
- Controls the event registration/generation process – callers can register to receive asynchronous notifications of changes to the services cache. The ServiceRegistrar maintains the list of event registrations (including the event generation criteria), each of which is on a time-based (lease) basis.

Without RMI, more of the operational load for Jini interactions is localized in the ServiceRegistrar object on the client devices; less work is done by the device hosting the LUS. Consequently, the LUS resource requirements are much reduced.

**VCSI and Service Discovery API**

Ford Research & Advanced Engineering created an in-vehicle software specification, in essence middleware, called Vehicle Consumer Services Interface (VCSI). VCSI is a software infrastructure that defines the interface between applications and the underlying in-vehicle hardware and software environment. This interface has the potential to become the de-facto a standard across the Ford enterprise, and in-vehicle applications are supposed to be written against this specification. VCSI includes a set of Application Programming Interfaces (APIs), such as Human Machine Interface, Personalization, Security, Policy, Communication, Media, Resource Management, etc., covering the wide range of multimedia, telematics, and infotronics applications. VCSI is designed using Java programming language and the OSGI (Open Services Gateway Initiative) framework.

Core to the fundamental concept of the project is that every aspect of it -- from data to applications -- is highly dynamic. Applications work with services that will be made available to them by VCSI. The notion of service is a logical one: services could be associated with devices (e.g. tuner or CD player), specific applications running on the device (e.g. phone's address book), some core vehicle functionality (engine speed information, climate control), or server side applications (entertainment content download, diagnostics). The services could physically reside locally in the vehicle, mostly factory installed, or remotely.

The notion of local services assumes their stability, since they are static relative to the vehicle. It is safe to assume that between the time the engine was turned off and on again, the availability of these services will not change. Remote services present a totally different picture, which can lead to very specific challenges. We cannot make any assumptions about the availability of such service in the vehicle at any given time. Therefore, we need a mechanism that at the very least will help to recognize at run time which services are presently available. This is a purpose of Service Discovery API.

Service Discovery API defines the interface that applications use to discover services that are available in the vehicle. The design of this interface does not make any assumption about how the service discovery is implemented. It could be based on basic service discovery capabilities provided by the OSGI framework, or use other more sophisticated mechanisms, such as Jini, UPnP, Salutation, or any combination of those. Therefore, special attention was given to the point that no implementation details should be accessible to applications. For example, OSGI specification, release 3.0 [11] defines the optional package (bundle) for Jini, and there are a few implementations of this specification on the market. The problem with this package that it assumes that the application will know that it has to deal with Jini to do a service discovery and therefore that it has to incorporate Jini-specific behavior.

Default implementation of the VCSI Service Discovery API is based on the OSGI 'native', registry-based, service lookup mechanism, which is quite limited. Jini-based implementation provides much richer tools for service discovery but poses a challenge of how to combine these two mechanisms without compromising the abstraction of applications from any knowledge about Jini. For example, Jini uses object-based services lookup based on assertions on values of attributes, while OSGI is limited to filtering based on primitive types. On the other hand, Jini does not provide for assertions, such as greater than or less than. To deal with this incompatibility, new classes were developed. These classes could be considered either as subsets or extensions of the Jini specification. On the one hand, they restrict the choice of objects used during a service registration, while at the same time they extend their behavior to provide for more flexible filtering.

Such approach does, however, limit somewhat the number of Jini-compliant devices that will be able to work with the vehicle. This limitation does not impede the capabilities of VCSI applications. After studying the subject of the integration of consumer devices in the vehicle, we introduced the assumption that we do not want the situation where ANY device or application coming from outside the vehicle will work in our vehicles. We want to make sure that unless the service provided by such devices or applications is trivial (and all security and policy constraints are preserved), we will have control over the usage of such device or application. In the case of a device, it means that we assumed from the beginning that it is possible (and in many cases desirable) to load a piece of software code specific to Ford into the consumer device. Most of the detailed reasons and consequences of such assumption (as well as the practical issues related to the implementation and maintenance of such solution) are outside of the scope of this paper. However, making such assumption from the beginning significantly simplified the task of creating the Jini-based implementation of VCSI Service Discovery API.

Another assumption that also positively affected such implementation is the static nature of LUS in the VCSI environment. Jini was designed to work in an 'ad-hoc' networking environment, in which there are no traditional servers, and services are totally standalone and independent. Under normal circumstances, the Jini client has to discover LUS using multicast or unicast discovery. Since we have full control over VCSI design and implementation, we can hardcode in the client the address of LUS (which is implemented by VCSI itself), thus bringing certain structure into this by definition unstructured environment. The device or application always knows how to locate LUS in VCSI-enabled vehicles, and it therefore no longer matters what kind of protocol the client uses to gain access to the LUS. Such simplification allows the extension of the boundary for the potential application of Jini technology. The process of LUS discovery defined in Jini specification is based on the TCP/IP protocol. Beyond that point any protocol could be used, so that in-vehicle modules and devices that do not contain TCP/IP implementation could also participate in Jini-based infrastructure.

VCSI was designed with the goal of having as small a footprint as possible. It is based on Java 2 Micro Edition (J2ME), Connected Device Configuration (CDC) and Foundation profile [12], [13], [14]. RMI is an optional package in this specification. This package is crucial for open distributed systems, but we decided early on that it cannot be used in VCSI due to resource constraints. The JMatos solution enabled VCSI to obtain all the flexibility and dynamism of Jini with very little overhead.

**Jini Without Java**

The use of Java on Ford’s VCSI platform provides great flexibility. Services, written in Java, may be obtained from other devices and run on the VCSI without the service author needing to know anything about the VCSI hardware and software details. It is assumed that devices offering services are Java-enabled.

There is a variety of deeply embedded computing devices in a typical automobile, many of which might offer useful services. However, unlike the VCSI or other larger platforms in the vehicle, these devices are extremely resource constrained. Moreover, current programming standards do no allow the use of Java on mission critical systems.

Under the assumption that such embedded devices have no interest in acting as Jini clients (i.e., they will not run other, Java-based services) but do want to offer a software-based service, PsiNaptic has rewritten JMatos in C, yielding CMatos. It is a Jini-compliant LUS implementation that offers static or dynamic services. Static services are created offline and bundled with the CMatos LUS software image on the embedded device. Dynamic services can be added to a CMatos LUS by a Java-enabled Jini client device.

The resulting implementation requires less than 60 kilobytes in total; that includes a simple RTOS, communications stack, and the LUS. Since there is no requirement for a Java virtual machine, CMatos can be fit into a single, low-end microcontroller.

With CMatos, any vehicle subsystem can offer services to the VCSI, or even through it to other devices. For example, smart sensors could be very good candidates to incorporate CMatos. Having a network of Jini-enabled smart sensors and actuators will provide for totally flexible, dynamic system where the software application does not have to be tied to the particular vehicle model or platform. This approach also extends the boundary of the potential usage of Jini technology in the vehicle since this technology could be leveraged in traditional 'core' vehicle applications. In a way this could be a real 'convergence' of the technology, originally created for consumer devices, and core automotive technologies.

## 1.2 Conclusion

Jini implements a distributed computing environment that can support the idea of 'connect anything, anytime, anywhere'. It enables devices and applications to discover and share services and information without the need for human configuration and management. The original implementation of Jini precluded its use in a resource-constrained environment, such as automotive. PsiNaptic's small footprint implementation of the Jini lookup service, called JMatos, eliminated obstacles related to the feasibility of using Jini in the vehicle.

Current in-vehicle Java based middleware developed by Ford Research & Advanced Engineering uses JMatos for its Service Discovery API reference implementation. Further changes were made based on some assumptions intended to simplify the discovery process and emphasize an abstraction of applications from the actual service discovery mechanism.

The CMatos (Jini without Java) solution brings the Jini idea to a whole new level in terms of in-vehicle use by potentially

going beyond infotainment applications.

## 1.3 References

[1] K. Arnold, et al, The Jini Specification, 2nd ed., Reading, Mass.: Addison-Wesley, 2001.

[2] W.K. Edwards, Core Jini, 2nd ed., Upper Saddle River, NJ: Prentice-Hall 2001.

[3] Sun Microsystems Inc., The Remote Method Invocation Specification, 2001.

[4] www.upnp.org/resources.htm/

[5] www.jxta.org

[6] C. Bettstetter, C. Renner, "A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol," Proceedings EUNICE 2000, Sixth EUNICE European Summer School, Twente, Netherlands, September, 2000.

[7] www.sun.com/jini/specs

[8] Vienna University of Technology, Department of Automation, "Jini Connectivity of Home and Building Networks" project; www.auto.tuwien.ac.at/jini/

[9] Rochester Institute of Technology, Information Technology Laboratory, "The Anhinga Project"; http://www.cs.rit.edu/~anhinga/

[10] http://developer.jini.org/exchange/projects/surrogate/

[11] http://www.osgi.org/

[12] http://java.sun.com/j2me/

[13] http://java.sun.com/products/cldc/

[14] http://java.sun.com/products/cdc/

Additional Sources

[1] S. Ilango Kumaran "Jini Technology. An Overview", 2002 Prentice Hall PTR

[2] W. Keith Edwards " Jini Example by Example", 2001 Prentice Hall PTR

[3] Hinkmond Wong "Delivering Jini Applications Using J2ME Technology", 2002 Pearson Education

---

[1]
JXTA, Jini and RMI are trademarks of Sun Microsystems Inc.

[2]
The fallacies, reworded as problems are the following; networks are unreliable, the latency of a network is variable, bandwidth isn't constant, networks are insecure, network topologies are variable, administration of networks (and applications) is not uniform, access and transportation costs are variable. See pp. 45-46 [1].

[3]
A detailed specification for the ServiceRegistrar and other aspects of the LUS can be found in the Jini specification [1].