Hybrid service-oriented architectures: a case-study in the automotive domain

Conference Paper · January 2005

DOI: 10.1145/1108473.1108487 · Source: DBLP

6 authors, including:

Luciano Baresi
Politecnico di Milano
281 PUBLICATIONS **7,754** CITATIONS
SEE PROFILE

Carlo Ghezzi
Politecnico di Milano
365 PUBLICATIONS **9,791** CITATIONS
SEE PROFILE

Antonio Miele
Politecnico di Milano
61 PUBLICATIONS **911** CITATIONS
SEE PROFILE

Matteo Miraz
Politecnico di Milano
14 PUBLICATIONS **200** CITATIONS
SEE PROFILE

Hybrid Service-Oriented Architectures: A Case-Study in the Automotive Domain

Luciano Baresi, Carlo Ghezzi, Antonio Miele,
Matteo Miraz, Andrea Naggi, Filippo Pacifici
Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza Leonardo da Vinci 32
I-20133 Milano, Italy

ghezzi@elet.polimi.it

ABSTRACT

Vehicles are becoming complex software systems with many components and services that need to be coordinated. Service oriented architectures can be used in this domain to support intra-vehicle, inter-vehicles, and vehicle-environment services. Such architectures can be deployed on different platforms, using different communication and coordination paradigms. We argue that practical solutions should be hybrid: they should integrate and support interoperability of different paradigms. We demonstrate the concept by integrating Jini, the service-oriented technology we used within the vehicle, and JXTA, the peer to peer infrastructure we used to support interaction with the environment through a gateway service, called J2J. Initial experience with J2J is illustrated.

1. INTRODUCTION

Vehicles are becoming complex software systems where many different components and services need to be properly coordinated [5]. Consider, for example, the large number of intra-vehicle features that must be controlled and coordinated, such as wheels, brakes, injection control, ABS system, and other subsystems and accessories, including climate control, HI-FI, and GPS. Furthermore, advances in networking technologies are now adding complexity (and new challenges), by allowing vehicles to communicate and coordinate their behaviors with the external environment and with the other vehicles in proximity. Vehicles may also be viewed as carriers of services, e.g., providing information on the traffic encountered during their journey to other vehicles, or interacting with the environment, which might provide location aware services to the people in the vehicle.

Both intra-vehicle, inter-vehicles, and vehicle-environment interactions require flexibility and, possibly, some degrees of self-configuration. Intra-vehicle (IV) interactions must deal with the many configurations of modern vehicles and also with the mobile devices that might be used in the car. Although components do not vary frequently, we cannot envisage software architectures where the communications among components are hard-wired. The different configurations would lead to a very complex architecture with many different variants.

Inter-vehicles and vehicle-environment (also called, extra-vehicle —EV) interactions consider the different vehicles (and the environment) as components of a dynamic federation, which supply and require services. Service suppliers and requestors are not fixed nor predefined, but can join and leave the federation dynamically. In this context, it is not only a problem of understanding who is "alive" out of a predefined set (i.e., the mandatory elements and additional components that can be operated on a vehicle), but available features can change with the context and thus need to be discovered, negotiated, and bound dynamically.

Dynamism, flexibility, and self-organization are the key features of *service-oriented architectures* (SoAs) [2]. A SoA is a software architecture where available services are advertised through brokers, clients can discover and negotiate them, and the binding between clients and providers can be set in flexible ways, e.g., dynamically. This solution works for both IV and EV interactions. In the first case, it helps discover the features installed in the vehicle and configure them. In the second case, since services change with the context, discovery and negotiation capabilities help identify available components and set proper federations.

The investigation of SoAs in the automotive domain — both at the IV and the EV level— is one of the main research goals of our participation to the EU research project called SeCSE (Service Centric System Engineering, [7]) whose goal —among others— is to investigate the use of SoAs in the automotive domain. In order to use it as a guideline for development, the general idea must be refined into detailed design and implementation guidelines. We argue that in doing so we do not force software architects to follow a uniform style and adopt a single middleware technology. Rather, we aim at designing *hybrid solutions*, where *services* are supplied through different interaction paradigms and middleware systems. For example, interactions within the vehicle and between vehicles and the environment may require different solutions. Within a vehicle, we can adopt a middleware solution that is based on some shared centralized components. As we move to EV features, we must foresee a decentralized organization where services are suitably replicated to improve their availability. Moreover, the degree of reliability is higher in the vehicle than in the open

SEM 2005 September 2005 Lisbon, Portugal

environment.

Another reason for dealing with integrating different solutions —both in terms of interaction paradigms and middleware technologies— is of practical nature. The IV and the EV worlds have different stakeholders and their services may be developed by different organizations. The need for integrating subsystems that follow different styles might be dictated by existing legacy components.

For all these reasons, we argue that hybrid SoAs should be investigated. A *hybrid architecture* is one that integrates different communication paradigms and technological platforms. Each element perceives a homogeneous federation where all the components communicate by using their primitives. Suitable gateways are then defined and implemented to connect the heterogenous parts and supply the required transformations.

In this paper, we report on our first experiences with a hybrid architecture that integrates Jini, which is used as service-oriented IV technology, and JXTA, the peer-to-peer infrastructure we have used for EV services. The interactions between Jini and JXTA are managed by J2J, the gateway we designed and implemented to deploy the necessary integration in the hybrid SOA. The choice of these technologies is simply motivated by their diffusion and by their suitability to support IV and EV interactions. They are the representatives of a wider class of middleware technologies, and the ideas behind J2J are not limited to this specific instantiation of *hybrid systems*, but have broader applicability.

J2J transforms the service invocations in Jini into messages in JXTA, and viceversa. It also allows components to dynamically identify the services they need and thus the components they want to interact with. Besides presenting the general concepts and the high-level architecture of J2J, we also exemplify the message flows when we make the environment (JXTA) communicate with the car (Jini) and the other way around. These examples demonstrate the capabilities of integrating two different technologies, but also two different interaction paradigms. The adoption of JXTA, and thus of a P2P approach, help solve the problems of the correct distribution of information in wide-area environments, where a centralized repository or coordination would not be feasible.

The rest of the paper is organized as follows. Section 2 briefly surveys the technologies used to conduct our experiments. Section 3 presents J2J, which is our proposal for a gateway that supports the cooperation between Jini and JXTA frameworks. Section 4 applies the proposed gateway to the interaction between a vehicle and the environment. Section 5 describes some related approaches to foster the cooperation among different communication frameworks and Section 6 concludes the paper.

2. TECHNOLOGIES

The proposed hybrid architectures are based on Java-related technologies, namely Jini, Javaspace (which is integrated in Jini), and JXTA.

Jini [9] introduces the concept of dynamic networking. It is a set of specifications that describe how to build adaptive distributed systems that run on the Java platform. A dynamic network is created with the assumption that the environment can evolve and the system has to adapt to the changes.

Jini Lookup Services (LUS) allow services to join the network by registering themselves with a LUS. Since a LUS is itself a Jini service, there can be multiple LUSs on the network and they can come and go as every dynamic Jini service.

A Jini component might know at design time the correct address of a LUS or discover it at runtime by sending a broadcast request for lookup services. If the Jini component already knows the correct address of a LUS, it contacts it directly, otherwise it contacts the first LUS that answers to its broadcast request. In both cases, the LUS sends a proxy to the Jini component that then uses it to lookup or register services in the network.

During the registration phase, the LUS and the Jini service agree on the terms of the lease, such as the duration of the lease, the objects exposed to the other services, and a set of optional attributes. Such exposed objects can be either copies of existing objects or proxies to let the client execute the functionality locally or remotely, respectively.

The optional attributes, set during the registration phase, can be used for the categorization of services, helping differentiate services that offer the same capabilities. Jini, in fact, offers the ability of searching for a service using its interface, and a set of attributes, to identify a specific service in a group of services that export the same interface.

Since Jini clients choose Jini services according to the Java interface they implement, they automatically use the service that provides the functionality they need without even knowing how it is called or where it is located. This makes service substitution simpler and more affordable: the client is able to recognize and use new service implementations as soon as they become available.

Javaspace [9] implements a Linda-like tuple space [3]. In a tuple space, different components can share information in a logical common space, as required in blackboard-based architectures. This space, called repository, contains the shared objects, called tuples, and offers the basic mechanisms for adding, reading, and extracting tuples.

Since it is implemented on top of Jini, Javaspace is a Jini service. Thus, when a component needs to use its features, it joins the Jini framework by searching for the Javaspace service in an appropriate LUS. Once it finds an active Javaspace, it uses its features: it can add, read, and extract tuples from the repository. Users can add objects to the Javaspace to share information: when a service needs to communicate with other services, it puts the relevant information in a tuple and adds it to the Javaspace. Each tuple is characterized by a special-purpose interface, according to the message type it belongs to. Services can look for the tuples stored in the Javaspace and filter the results with respect to a particular type of tuples. The searching mechanism is based on pattern matching of the tuples' interfaces and optionally on some attributes. Once a tuple is found, the caller can read it or read and remove it atomically.

Javaspace can be used to build a dynamic framework of decoupled services. All services can cooperate with the others through message exchange. The collaboration is not based on the knowledge of involved services. Each service needs only to know how to access the Javaspace and how to interpret stored messages (tuples).

JXTA [11] is a set of protocols that allow devices to communicate in a peer to peer (P2P) way. JXTA peers create a virtual network where any peer can interact and communicate directly with the others. JXTA is designed to build

frameworks which are highly dynamic and scalable. The main components of JXTA are its peers and the communication channels used to allow the peers to interact. A JXTA framework can work both without a centralized authority and with some peers that act as lookup services to improve network performance.

JXTA provides several features that support the organization of peers and the discovery of services. Groups of peers can also be related to each other by some special-purpose criteria to create a hierarchical structure among groups. When a group is created, it informs the other peers about its existence. From then on, existing peers can join the new group. Notice that a peer can join different groups. Advertisements are fragments of XML code that identify an entity, i.e., a peer, a group, or a service, in a JXTA framework. They contain a description of the object and the information needed to reach it. They are created at the same time as the entity, and they are spread to the other peers. The discovery service is an infrastructure that permits the discovery of a specific JXTA entity. It can find advertisements published by other entities in the same group of the searching peer.

When a peer knows another peer, it can instantiate a communication channel with that peer and they can exchange typed objects serialized via XML messages. Pipes are the high-level abstraction for communication channels. A pipe is a link between two or more peers that provides primitives for transferring typed objects. Every pipe has its own advertisement and a peer can bind itself to a given pipe to transmit data.

3. J2J

The goal of J2J is to create hybrid architectures to enable a bi-directional communication between a Jini network and a JXTA network (Figure 1). We can also replicate J2J to set hybrid architectures with one JXTA network and several JINI networks: each J2J gateway enables a Jini network to communicate with the JXTA network. Every bridge can operate in a totally independent way with respect to the others. Every Jini network uses the same cooperation paradigm to communicate with JXTA; the same applies when JXTA communicates with a Jini network.

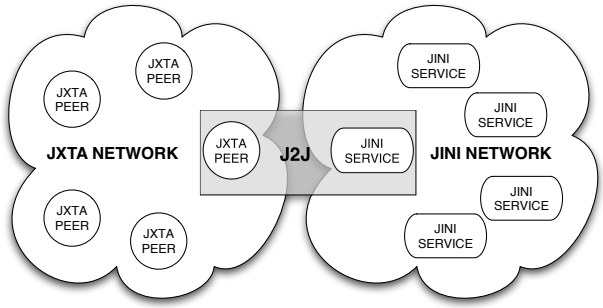

Figure 1: J2J hybrid architecture.

Before explaining the details of J2J, we must understand that Jini and JXTA use different communication paradigms and thus their integration implies the capability of mimicking the Jini communication paradigm in JXTA and viceversa.

Jini supports synchronous communications, where the caller is blocked until the callee does not answer. In contrast, JXTA supports asynchronous communications, where the caller is not blocked after performing a call and it is free to continue its execution. If needed, the callee can contact the caller to send a response message.

A second difference concerns the collaboration schema. Jini can support both direct invocations, through remote method calls, and blackboard-based interactions through Javaspace. In the first case, the caller knows exactly how to contact the callee and the communication takes place directly with the other party. In the second case, the caller does not know exactly how to contact the callee, so it uses a blackboard to communicate: the caller sends a message to the blackboard, and then the possible recipients read the message posted on it. One or more components may decide that they are interested in the message posted on the blackboard and they can react consequently.

JXTA only adopts the direct communication schema, but since our goal is to integrate the two frameworks, we must merge the two communication schemas. J2J allows Jini to borrow asynchronous communications from JXTA, and JXTA to inherit synchronous and blackboard-based communications from Jini.

3.1 Message types

J2J allows a JXTA network and a Jini network to exchange the following messages. If we consider the information flow from JXTA to Jini, J2J supports:

- **Direct messages.** The sender, a JXTA peer, knows exactly the kind of services that are supposed to receive the message. It knows that the receiver is a Jini service that implements a specific interface. The sender specifies the parameters to lookup the service, the specific method it wants to invoke, its actual parameters, and if it wants to be notified about the results of the method execution or about possibly thrown exceptions.

- **Indirect messages.** In blackboard-based collaborations, the sender (JXTA) sends a message without any addressing information. J2J receives the message, recognizes that it is a blackboard message, converts it into a tuple, and stores it in the Javaspace. Interested Jini services are notified and can use the tuple.

If we consider the information flow from Jini to JXTA, J2J supports:

- **Direct messages.** Jini services call a proper operation (specified later) on the Jini service published by J2J, with the message to be sent and the addressee as parameters. J2J builds a JXTA message and sends it to the JXTA destination. If no suitable peers exist, this exception is sent back as separate message.

- **Indirect messages.** In the blackboard model, Jini services store their messages (tuples) in the Javaspace, as any Jini service would do. J2J is notified of these added data, discovers if there is an external JXTA group that wants to receive them, and forwards them to the group. To make it possible, J2J uses the *Directory Service* to find if there are peers interested in the tuple and to retrieve their addresses.

J2J relies on the messages exchanged between the two parties, and does not address QoS-related issues. For the sake of simplicity, we assume that the meaning of a message is the same in the whole system. We do not consider ontologies and semantic translations, but we rely on standardized messages.

3.2 J2J architecture

J2J is seen as a Jini service within the Jini network, while it becomes a peer in JXTA. Figure 2 shows the architecture of J2J, where we have a *JXTA Module* and a *Jini Module*, along with their interfaces called *JXTA Interface* and *Jini Interface*, and a *Directory Service*. Moreover, the *Jini Module* uses a dedicated interface to communicates with the Javaspace.

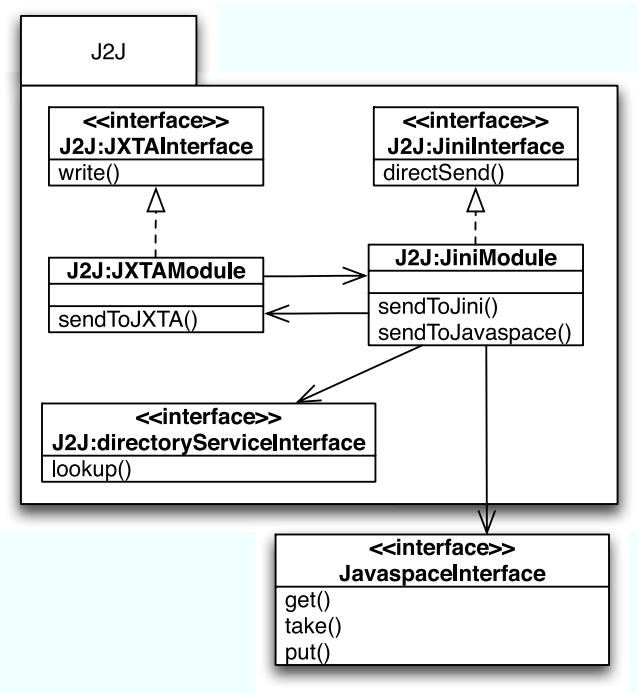

Figure 2: J2J architecture.

JXTA Module

The *JXTA Module* implements the JXTA peer used by J2J to communicate with the JXTA network. When J2J wants to communicate with a specified JXTA peer, it joins one of the groups containing the destination peer. The groups to which J2J subscribes depend on the particular context in which J2J is used and are not hard-coded in the component. The setup phase follows the same policies chosen to configure the Jini network.

We assume that the messages that come from the JXTA network are standardized and we distinguish between direct and blackboard-based communication. As to direct communication, messages contain the information used by the Jini LUS to find the specific destination service, the name of the method to be called, and the parameters that have to be used. In contrast, the structure of blackboard-based messages is not fixed by J2J. This distinction comes from the fact that, in direct communications, the caller knows exactly the interface of the callee, while in indirect communications the caller only deposits some data in a shared space and knows nothing about the possible users of those data.

The *JXTA Module* implements a `write` method that registers a JXTA handler for every JXTA group J2J is listening to. The handlers process JXTA messages as soon as they arrive. The method distinguishes between direct and indirect messages —since the difference is in the content of the XML message— and uses an external library to build an object that corresponds to the XML message.

If we consider indirect messages, the *JXTA Module* simply calls the *Jini module* by passing the object created above, but without specifying the destination. The message is stored in the Javaspace. If we consider direct messages, the object also contains the destination of the message (i.e., the name of the service for the LUS of the Jini platform). This module is also responsible for mimicking synchronous interactions with JXTA peers. It keeps track of those peers that have sent a direct message and want a response, and sends appropriate messages as soon as the responses arrive from Jini. It also sends to JXTA fault messages if an error occurs while calling the Jini service.

The *JXTA module* exports method `sendToJXTA` to the Jini module. This method is used to send messages to the JXTA network. It takes as parameters the message (as a Java object) and the destination (as a hierarchical structure representing the path groups to a JXTA peer). This method joins the specified group, serializes the message as an XML document, and sends the document either to the single specified peer or to the whole group. Since the *Jini Module* assumes synchronous communications, if the message requires a response, this module also transforms the response message for the JXTA peer into the correct response to the Jini invocation, otherwise the call to `sendToJXTA` returns immediately. If a response is required, but it does not arrive within a specified amount of time, the method raises an exception.

Jini Module

The *Jini Module* implements a Jini service registered in the Jini network. This enables J2J to communicate with the Jini network.

When this module receives a call from a service within the Jini network, it identifies the requested JXTA group and calls the *JXTA Module* to forward the message to the interested peer. Method `directSend`, which requires two parameters: the message to be sent and a recipient address (i.e., the necessary path to reach a peer in the JXTA network) calls method `sendToJXTA` of the *JXTA Module*, and also handles responses and error messages from JXTA. This method cannot return before delivering such messages (return values or exceptions).

The same approach can be used to make a Jini service communicate with other Jini networks. J2J allows the Jini service to interact with them as if they were JXTA peers.

When the *JXTA Module* sends a message to the *Jini Module*, we identify the involved service and pass the invocation to it. This is done through method `sendToJini`, which uses the object containing the message and the destination for the lookup service (i.e., the interface of the service) as parameters. This method returns after the Jini service has returned. If there is a problem with the invocation of the Jini destination service, `sendToJini` raises an exception.

Besides supplying the functionality to manage the ser-

Figure 3: Working context

vices in the Jini network (such as registering to the LUS and updating the registration), this module also manages the blackboard system (i.e., the Javaspace). It allows the *JXTA Module* to put data (messages) on the blackboard, and reads those tuples that have to be sent to the JXTA network.

Directory Service

The *Directory Service* contains pairs of message types and JXTA group paths. It helps the *Jini Module* find the destination of indirect messages from Jini services by providing a way to associate the type of a message with the JXTA groups interested in it. As already described, messages stored in the blackboard by Jini services can be sent to the JXTA network. Since the *Jini Module* is a Jini service, it is triggered by the events that correspond to changes in the Javaspace, and as soon as it understands that a new tuple (message) is available, it uses the *Directory Service* to discover if there are JXTA peers that are interested in that information. This means that the *Directory Service* offers a method to find the destination by supplying a message type, but also ways to allow JXTA groups to register their interest in particular message types.

4. CASE STUDY

Figure 3 shows a screenshot of the prototype environment in which J2J works. The scenario comprises two main subsystems: a vehicle and the external environment (simulated on a local-area network).

The vehicle contains J2J and a Jini network (i.e., all the services inside the car are Jini services); the external context is realized by a JXTA network (i.e., all the services in the external context are JXTA peers).

IV services comprise, for example, onboard displays, the engine processing unit, a GPS receiver, a mobile phone, and the infotainment system. EV services comprise traffic information services, intelligent street signals, other cars, public assistance vehicles, and automatic toll systems.

J2J enables the cooperation between these two families of services and is the key component to set the hybrid architecture. J2J allows a service registered on one side to cooperate with a service registered on the other side. It manages the communications between the inside and the outside of the vehicle as described in the previous sections, interprets service calls, and redirects them.

Given this context, a first scenario (Figure 4) shows the case of a message flow from the environment to the vehicle. The car arrives close to a tunnel; a sensor in the JXTA network sends to all the cars near the tunnel a message to signal the entrance. The car receives the message and displays it on the on-board display, turns the lights on, and the radio off: these are all Jini services.

More specifically, the tunnel, which is a JXTA peer, sends a message to the car, exposed by J2J as another JXTA peer. The message, which is broadcast to the group the car is registered to, is received by the J2J's *JXTA Module* that creates the object with the request and passes it to the *Jini Module*. Since the tunnel does not know the intra-vehicle components interested in the notification, it sends an indirect message and thus the *Jini Module* puts the message in the Javaspace by passing the object created by *JXTA Module*. The *Lights*, *Radio*, and *Display* are notified by the Javaspace: they lis-

ten to the Javaspace for a specified type of objects, which in this case is what sent by the tunnel, and react accordingly.

In the second scenario (Figure 5), we show an example of message flow from the vehicle to the environment. If we suppose that the car engine breaks down, the appropriate Jini service must communicate the fault to the car service center, which is a JXTA service.

Again, the *Engine*, a Jini service, builds an *engine fault* message (which is a Java object), and invokes the Javaspace to store it. The Javaspace notifies both the *Warning Light* in the car and the *Jini Module*. Both these components are Jini services. J2J uses the *Directory Service* to find the right JXTA destination by passing the type of the message object. The *Directory Service* replies with the path of the JXTA destination group. Now, the *Jini Module* invokes the *JXTA Module* by passing the message object and the path to the destination. This module subscribes to the right group (if it exists), serializes the object into an XML structure, and broadcasts it to the specified group (which contains the *Car Service Center*). In this case, J2J lets the car see the external component as if it were a Jini service.

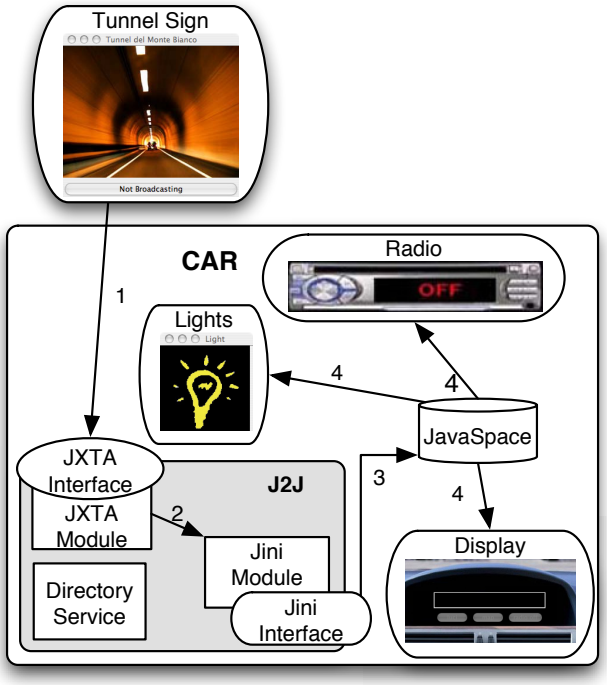

Figure 4: From environment to vehicle

In the third scenario (Figure 6), we have two cars that share traffic information without using a centralized exchange entity. We can obtain such an interaction by making two Jini networks, i.e., the two cars, communicate directly through the JXTA network. We want to achieve this goal with a direct communication paradigm, i.e., without using the Javaspace. On each car, there is a Jini service that can report on traffic conditions and can also get the position of the car, maybe by using another Jini service that embeds a GPS receiver. When the service detects certain traffic conditions, such as a traffic jam, it sends this information to the JXTA network and thus to the other vehicle via J2J. It is also able to receive traffic information from other vehicles and present

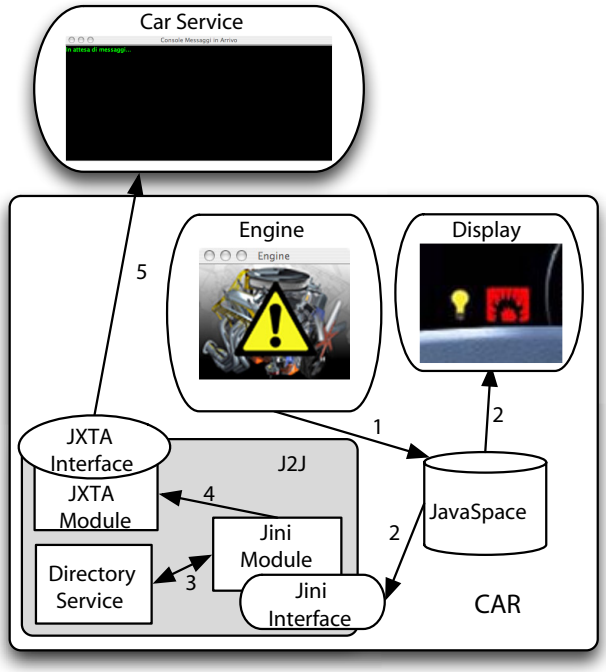

Figure 5: From vehicle to environment

them to the driver through some other Jini services. To enable direct messages, this service is supposed to have the same interface and the same lookup information on all the vehicles. The communication starts from the service located in a particular car, and the J2J in that car, which receives the message, forwards it to the interested JXTA group, so that every other J2J, which has joined the group, receives the message and forwards it to the Jini service on the car.

5. RELATED WORK

The integration of different middleware technologies is a well-known problem already tackled by several initiatives. In this short section, we do not want to address the general problem, but we prefer to concentrate on those proposals that resemble our approach.

There are a couple of attempts that aim at integrating JXTA with other service-oriented technologies. The *sourceforge* project called JXTA & Web Services Gateway project [10] is one of the proposals that are closer to J2J. It proposes a solution to connect P2P JXTA elements and Web services. The gateway advertises services between JXTA and Web Services environments and transforms the communication protocols: peers in JXTA and Web Services environments communicate by using SOAP/JXTA and SOAP/-HTTP transformations. Unfortunately, the project does not seem to be active anymore. Also the JXTA SOAP package [4] provides an integration point between JXTA P2P networks and Web services and allows the same code to be used as a JXTA P2P service and as a conventional Web service. They do not supply a stand-alone component to make the two middleware systems interoperate, but this package allows designers to "wrap" their code and see it as an element of either framework.

The proposal in this paper imposes a mention also to

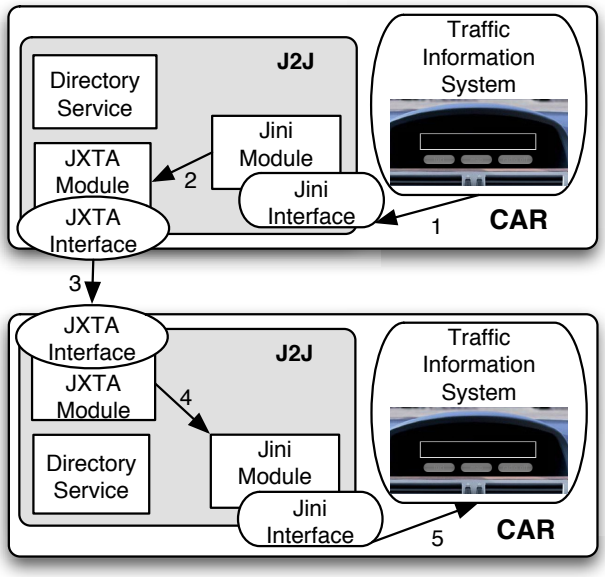

Figure 6: from vehicle to vehicle

the Jini car developed by Sun [8], while the service-oriented paradigm motivates two further references: OSGi [6], which is another example of service-oriented middleware often used in the automotive domain, and AMI-C [1], which is an initiative to standardize the messages exchanged among the components in a car.

6. CONCLUSIONS AND FUTURE WORK

The paper presents our first experiments with J2J to implement hybrid architectures and demonstrate them in the automotive domain. A hybrid architecture is an architecture where different portions use different coordination and communication paradigms to supply *services* to the other portions. J2J is the gateway provided to make the different portions interoperate.

The experiments conducted so far, and partially presented in this paper, have given interesting results and are encouraging us to implement other plug-ins and conceive more heterogenous systems to let components communicate in fully heterogeneous environments.

This implementation of J2J is based on JXTA and Jini, but we have plans to re-implement it using other service-oriented technologies: the main principles described in this paper would remain untouched; we should only tailor the implementation of the dedicated modules on the chosen infrastructures. As shown in Figure 1, the architecture is composed of loosely-coupled modules that can be replaced by new ones. For instance, if we wanted to make Jini communicate with a Web Services, it is enough to replace the *JXTA Module* with a new *Web Services Module* with the same functionality (i.e., the same interface). These implementations can also be extended by adding features that do not concern with the communication paradigm, but address QoS issues, like performance, trustworthiness, and security.

Our future plans include a more detailed assessment of J2J by both selecting other case studies and extending the set of supported frameworks. This also means better policies to select available services and the capabilities of on-the-fly negotiation of QoS parameters. More generally, we will continue to investigate SoAs and their applicability to important practical areas, such as the automotive domain. This will be part of our contributions to the SeCSE project.

7. REFERENCES

[1] AMIC Consortium. AMIC - Automotive Multimedia Interface Collaboration. `www.ami-c.org/`.

[2] D. Booth, H. Haas, and F. McCabe. *Web Services Architecture*, 2004. `www.w3.org/TR/ws-arch/`.

[3] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[4] JXTA org. JXTA SOAP Project home page, 2004. `soap.jxta.org/`.

[5] I. Krueger. Researcher in Focus. `www.calit2.net/researchers/krueger/index.html`.

[6] OSGi Alliance. Osgi web page. `www.osgi.org/`.

[7] SeCSE Consortium. Project web page. `secse.eng.it/`.

[8] Sun Microsystems. The Network Is the Car, 1999. `java.sun.com/features/1999/06/concept_car.html`.

[9] Sun Microsystems. Jini Network Technology, 2004. `wwws.sun.com/software/jini/`.

[10] The Gateway Development Team. JXTA & Web Services Gateway Project, 2004. `sourceforge.net/projects/j-x-w-s--gw/`.

[11] B. J. Wilson. *JXTA*. New Riders Publishing, first edition, 2002.