Enhancing JINI to support non–JAVA appliances

S. Müller–Wilken, W. Lamersdorf
University of Hamburg, Computer Science Dept.
Distributed Systems Group
Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
E-mail: {smueller|lamersd}@informatik.uni-hamburg.de

Abstract

With their introduction of the *JAVA Intelligent Network Infrastructure*, more commonly referred to as *JINI*, SUN Microsystems has presented a new and versatile way to design distributed systems. Based upon the consequent enforcement of only a small number of design principles (here especially discovery, lookup and leasing), JINI gives the means to create flexible, dynamic and robust application scenarios. Still, there is a huge drawback from the application designer's point of view: So far, JINI and its core component, the Lookup Service, are based on the assumption that all participants of a JINI scenario are JAVA–enabled. Especially with mobile system applications, this won't be realistic for some time to come and as even non–JAVA parties will gain a lot from being part of JINI scenarios, it is necessary to find enhancements to make them do so. This article describes a generic strategy to integrate non–JAVA parties into the JINI architecture, which provides a possibility to integrate arbitrary non–JAVA devices into the JINI community.

Keywords: middleware computing, ad–hoc computing, JINI, non–JAVA services, service integration.

1 Introduction

When the JAVA Intelligent Network Infrastructure [3] was revealed in early 1999, it meant a consequent step in achieving SUN Microsystems' goal of a thoroughly networked world as depicted in their slogan *The network is the computer*. While the building blocks JINI was constructed from (leasing, distributed events, a central lookup service, code migration, etc.) were not completely new to the field of distributed systems, their consequent enforcement within a computing middleware was. SUN Microsystems clearly predicted the low–cost consumer electronics market as a core segment for JINI, and thus appropriate means have to be provided to integrate devices as well, that are not capable of running virtual machines and do not offer the amount of resources an average JAVA application requires. SUN briefly sketched a few possible solutions to the problem in the JINI documentation [4] and presumes that no direct communication between a non–JAVA device and the environment will take place, but will rather be redirected through hosting proxy services.

We argue that this approach is not truly satisfactory and that especially with applications such as dynamic device drivers, devices, while not being JAVA–capable can still gain a lot from directly interacting with JINI's core components. We consequently introduce enhancements to the JINI architecture that make this possible.

The following section 2 will give an overview to the JINI architecture, its design principles and the core components that make up a JINI environment. Section 3 provides details on how SUN currently offers non–JAVA integration within JINI while section 4 enlarges on the JINI bootstrapping process, which is of utmost importance to any integration strategy. Section 5 introduces our approach to enabling JINI to support non–JAVA parties as direct clients to its core components. Section 6 draws an application scenario and identifies the advantages that arise from enhancing JINI to directly support non–JAVA parties. The last section will finally give a summary on the results so far and further enhancements to be made.

2 The JAVA Intelligent Network Architecture

Distributed application design has been a tedious task for long years, because it involved writing certain parts of a software over and over again, each time to address the special requirements of a platform. Not only differences in the operating systems in use, but often also subtle differences in data representation made this difficult and last not least error prone. And with each new platform entering the market, application developers had to go through the whole procedure again. It was the concept of middleware systems [1] introduced in the late eighties, that brought a solution. Middleware introduced an abstraction layer between application and operating system that hid most details specific to a certain platform behind standard call interfaces a developer could use. Since then, several middleware architectures have been developed, often adhering to popular design paradigms of their time and stressing certain aspects of distributed systems design: From first client–/server variants (DCE, AnsaWARE), over object oriented solutions (mainly CORBA) to mobile–agent systems like Voyager or Aglets some were more advanced with respect to security, some more performant etc.

The Java Intelligent Network Architecture was introduced by SUN Microsystems early last year as a JAVA–only middleware. From its basic design, JINI follows the client–/server paradigm. It is built around a registration service the Lookup Service (LUS), a kind of "yellow pages" directory which is used as a central repository for references to the service instances available in a JINI environment. The Lookup Service holds information on the service´s unique identifier, and the call interface the service is offering. Each offer can additionally hold attributes to further describe the service, e.g. the cost, capacity, speed etc. The most remarkable part of each registered service offer is the fact that it – in contrast to more traditional "yellow pages" services – also contains the actual code necessary for a client to contact the service. It is basically its extensive use of code migration that makes JINI unique amongst client–/server middleware platforms. Facilitated by JAVA´s byte code technology, clients in JINI not only get information on the interface a service offers but also a piece of code, the so called proxy, that implements the actual communication between client and server. Provided that the interface is known to – and understood by – the client, the server can change the communication primitives any time and without further notice to the client – as long as the client holds the appropriate proxy to use. And this again is assured by JINI´s second important design principle, the stringent use of time–bound resource leasing.

The use of lookup services is not new and has been thoroughly researched in the past years [6] [5]. Efficient trading mechanisms have been proposed and have led to various lookup and trading service implementations. The one big disadvantage so far was the fact that distributed systems are not fixed and a client thus cannot rely on a service being present if accessed, and not all services have the time to cleanly remove their advertisement before going off service.

Both can consequently lead to a situation where lookup services will no longer accurately reflect the situation within the environment and clients can no longer rely on the offers returned from calls to the lookup service. As a solution to this problem, JINI is inherently based on a leasing mechanism, that underlies all relationships within a JINI environment. Whenever someone offers a service within the JINI environment, advertises it to the public and enters a use–relation with a client, the relation is "timed". A lease will be created that holds the amount of time the server is willing to guarantee the service. When a lease comes close to its end, the relation has to be renewed by the client and the server can decide if it is willing to grant another lease. If it is not, the client will be informed and will have to ask the Lookup Service for another service offer. As this mechanism is a basis for all client–/server relationships in JINI, it also applies for all access to the lookup service. Consequently, service registrations have a lifetime and will be purged from the service directory if not refreshed by the server (as would be the case for a 'dead' service).

A third important aspect of JINI is the discovery mechanism in use. Discovery solves the problem clients and servers face at startup on how to find possible Lookup Service instances to direct their inquiries to. As one of SUN Microsystems' main aims with JINI was to provide support for highly dynamic environments, the "bootstrapping" process was a very important aspect to consider. In JINI, services by default use a multicast mechanism to find instances of the Lookup Service. They send their request to a well–known multicast channel each JINI Lookup Services registers with at its startup and have all Lookup Service instances reachable at that time send their details in return. They can then directly access one of the returned Lookup Service instances. The bootstrapping process can be reinitiated anytime a JINI participant fails to access a Lookup Service to help him back into the JINI community. While JINI also provides the means to permanently configure Lookup Service addresses into the JINI participants, it is this multicast–based mechanism of locating Lookup Services without prior configuration that gives JINI its flexibility.

The JINI bootstrapping process

Before enlarging on the details of our approach, we first will describe the two phases a JINI compliant device will pass when integrating itself into a JINI community. This will help to better understand certain design decisions we have made.

Within the initialisation phase, a JINI compliant device has two alternatives in advertising its service to the community, depending on wether it has been configured to use a specific Lookup Service instances (unicast discovery protocol) or rather to "find out" on the available Lookup Services at runtime (multicast discovery protocol). As the second case will be the more common, we will primarily refer to the multicast scenario.

- **discovery phase** — When attempting to enter a JINI community, the device will send out a discovery message to a well–known (to the JINI environment) multicast address/port combination. Within this message, it will send the protocol version it uses, the port on which it will listen for replies, the JINI groups it is interested in and (for efficiency reasons) the Lookup Service instances it already knows about [2]. Interested Lookup Services can respond to this request with a serialised service proxy object they send via RMI to the address/port combination they have identified as the source of the multicast

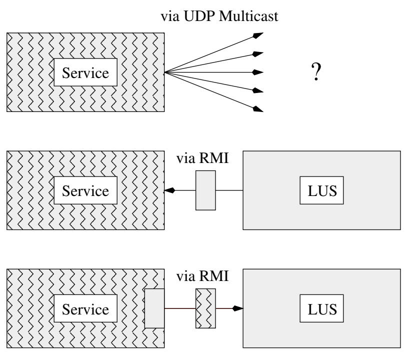

Figure 1. The JINI Discovery/Join Process

request packet.

- **announcement phase** — With the appropriate service proxy at hand, the JINI compliant device can then advertise its own service to the corresponding Lookup Service instance. To do so, it will build a registration object that contains its own serialised service proxy, a set of attributes that help to further describe the service offer and (if already available) its own unique identification string. This ID, comparable to a network card's MAC address, will be created by the first Lookup Service instance it ever registers with and will be returned as result of the registration process. It will then be used for all subsequent registrations. The registration is done through the service proxy the JINI compliant device has acquired in the preceding step and, again, the communication takes place via RMI calls.

It is important to notice that the two stages mix different levels of abstraction with respect to the communication primitive in use. The discovery phase uses plain UDP multicast for the request and RMI for the reply coming from the Lookup Service. The announcement on the other hand is JAVA RMI to its core. Starting from the construction of the registration object to its transmission back to the requester, all data is wrapped in JAVA objects and transferred via remote method calls.

It is primarily the fact that RMI is used within the discovery/announcement protocol that leads to a situation where non–JAVA appliances currently cannot participate in JINI environments without the helper constructions mentioned in the previous section. Still, we can see no reason that the use of high level communication protocols is mandatory for a successful bootstrapping procedure. Similar to using low level IP protocols for the discovery, low level IP protocols could be used for the announcement, as well.

3 Conception of the extended JINI–Architecture

3.1 Integrating non–JAVA parties into the JINI architecture

SUN Microsystems has foreseen the requirement to integrate non JAVA–enabled devices into JINI environments. In their "JINI Device Architecture Specification" cc [4] they formulate a set of characteristics a service needs to show to be JINI–compliant and introduce a number of means to integrate devices into a JINI community.

First of all, JINI requires the service *"... to be defined in terms of a datatype for the JAVA programming language ... "*. This basically means that any service will have to implement an interface that can be instantiated as part of a proxy and can be used by a JINI client application. A client will access the JINI service through the interface the proxy offers without further notice of how exactly the interaction with the actual service provider looks like. Additionally, the specification states that any JINI service will have to take place in the JINI discovery protocol to find available Lookup Service instances. JINI–enabled services will also have to be able to construct registration objects to transfer to Lookup Service instances they want to register with. They will finally have to be able to handle JINI leasing to manage registrations they hold at lookup services it has registered with.

As described in [4], SUN Microsystems basically sees three different strategies to integrate non–JAVA services into a JINI community that cope with the requirements stated above:

- **use of specialised virtual machines**
  With appliances that lack the necessary power to host a complete virtual machine, it is possible to reduce the VM to all that is required to participate in the discovery and registration steps, just to advertise its service proxy and have it follow the leasing protocol as expected. While this would impose the least restrictions with respect to its role within a JINI community, it still can cost much more resources than a small, inexpensive device could possibly offer.

- **cluster devices with a shared virtual machine as provided by a device bay**
  With this alternative, multiple devices can share the "cost" of providing the resources necessary for their participation in a JINI environment. According to SUN, devices in this scenario would be docked in a "device bay" and interact with it in some private protocol to announce their existence. The device bay could then advertise a previously stored service proxy at Lookup Service instances within its reach and subsequently act on behalf of the docked device. While this approach can help reduce the cost with devices that usually come in multiple instances (such as disks in a RAID array or light switches in a house), it would not make sense with devices that will rather not be stacked (such as PDAs or printers in an office building).

- **cluster devices with a shared virtual machine accessible via networks**

  This approach can be seen as a variation of clustering technique described before. Again, multiple devices share a common virtual machine. But while the devices in the previous approach shared a private protocol with the device bay they were physically collocated to, here they base the protocol on TCP/IP and transmit messages via networks just as any "full flavoured" JINI device would. While this will give more freedom in where to locate the devices within a network and more efficient means to share a common virtual machine, it comes at the cost of having to provide that additional part of the infrastructure. Especially with small or rather isolated JINI environments (as in home environments or ad–hoc networks), this might not be acceptable with respect to cost or the administrative overhead involved.

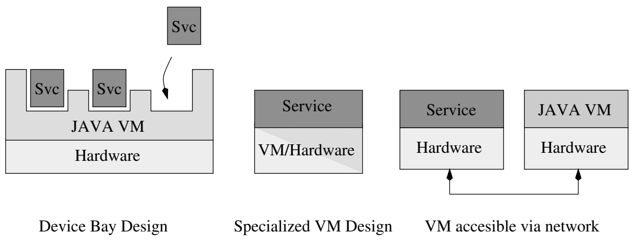

Figure 2. SUN's proposals for non-JAVA device integration

However, that there is one alternative still missing so far. In addition to the approaches presented above, one should think of appliances that on one hand don't have the resources to host a complete virtual machine to be fully JINI compliant, on the other hand are not commonly used in environments, where the administrative overhead implied in providing shared VM space and still would gain from participating in JINI environments. Examples for this class of devices might be small printers, measuring devices, lightweight PDAs, just to name a few. To these devices,

- communication between them and their peers in a private protocol which is not based on higher level JAVA protocols like RMI would suffice,
- a more than mechanical participation within the JINI environment (involving more than just "bootstrapping" themselves into the community and following a basic leasing protocol) would not be necessary,
- and the active use of the additional services a JINI can offer not be required.

Still acting like JINI devices would help them to be more manageable, more flexibly usable in evolving environments and therefore help them in being more attractive to potential customers.

3.2 A modified protocol for the JINI discovery process

At least, the following aspects have to be obeyed to integrate a non–JAVA service into a JINI environment with no further interaction of third party JAVA proxies:

- the Lookup Service has to be modified to be able to follow a modified protocol that relies on low level IP communication for the discovery/announcement process
- non–JAVA parties have to be provided with pre–compiled proxy objects they can send to the Lookup Service and that enable JINI participants to talk to them in a private, non–RMI protocol
- non–JAVA parties have to be provided with pre–assembled registration objects that can be transferred to the Lookup Service as part of the announcement protocol.

Through the use of protocol versions, JINI already inherently supports protocol variations. Modified discovery/announcement protocols can consequently exist in parallel to the one currently used by the JINI architecture. Each time, a protocol message is exchanged between a JINI enabled party and the Lookup Service, a protocol version number is transmitted. Through this number (that currently is defaulting to the value '1.0') the Lookup Service can decide if it is capable to interact with the opposite. We propose an enhanced discovery/announcement protocol that no longer forces parties to use RMI for the procedure, and using JINI's protocol versioning feature, we can easily integrate it into existing JINI environments.

When a non–JAVA party sends out a multicast request package, it uses a new protocol version number. When a modified Lookup Service catches a discovery package with this number, in contrast to sending a service proxy via RMI, it will open a TCP socket and send the address and port it is listening on back to the requester. The requester can then send the pre–compiled service proxy as part of the pre–compiled registration object to the Lookup Service instance.

The socket will be left open as a communication channel between the non–JAVA party and the Lookup Service. All management information (service withdrawal, lease renewal requests and so on) will take place through this channel and make further RMI communication between the Lookup Service and the non–JAVA party unnecessary.

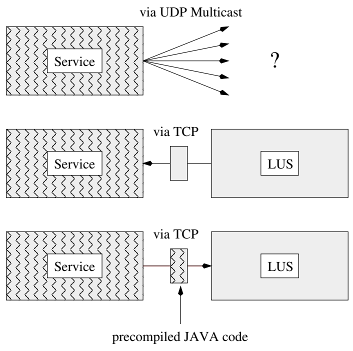

Figure 3. A modified Discovery/Join Protocol

One important aspect of using sockets to realise communication between the peers is that one has to have a structured mechanism to transfer information between them. Using a call interface requires some sort of Remote Procedure Call mechanism, which will most likely not be available beforehand. Adding an RPC implementation to the non–JAVA party will often fail due to limitations in storage capacity. Our solution is to use a lightweight protocol comparable to FTP or HTTP, based on standardized strings being exchanged between the peers. While this can be easily implemented even for small devices, it still has enough power to express even more complex protocols such as the leasing protocol used within JINI. The following section will give further details on how to handle leasing through this protocol.

3.3 Handling JINI Leasing

As described before, the leasing mechanism is of vital importance to the JINI architecture. Any modification to the JINI protocols will only be complete when integrating leasing and its protocols. Unfortunately, as JINI's leasing protocol is inherently based on JAVA technologies, it is much harder to 'tweak' it into non–JAVA parties than to just rely on proxies to do the leasing on behalf of the non–JAVA party. Still, when attempting to free non–JAVA devices from requiring external support, there is a solution available.

In its simplest form, JINI participants that support leasing merely use a proxy that refreshes their leases every time they are coming due. They do not implement an own leasing strategy themselves but renew a lease rather mechanically as long as this is possible. This regular renewal can be taken as a “heartbeat” by both the lease grantor and the leasing party and be used as a signal to discard the resource if the lease renewal should eventually fail. The simple “heartbeat” keeps a JINI environment free from invalid references and thus makes it more robust to failure. While the simple renewal strategy is currently implemented through JAVA proxies a JINI participant instantiates at runtime, it can also be realised through a standardized socket interface. This approach can then easily be followed by any non–JAVA party.

Whenever a non–JAVA party registers with a modified Lookup Service, a socket is left open between both communication partners (see previous section). As with a common JINI registration object, the client can fetch information (such as the lease duration) on the lease through this socket. It will also be able to initiate lease renewal requests through it. As with any other registration, the Lookup Service will decide if a registration is still valid by sensing if the lease is no longer renewed by the JINI party. As with the basic registration procedure, the actual leasing protocol is again realised through messages being sent between the resource grantor and the resource user.

4 Prototype implementation

We have implemented the concept described above based on JINI version 1.0, using a combination of ANSI C and PThreads to realise the non–JAVA service implementation. As mentioned above, we introduced a new LUS protocol version and modified the existing Lookup Service to correctly follow the socket based communication with its non–JAVA clients. To test our implementation in a more realistic setting, we have created a surveillance scenario in which a JINI based monitoring system is used to coordinate and integrate numerous small measuring devices that can be distributed in a building. The devices are simply attached to the local network, continuously read values from sensors attached to an integrated A/D converter and offer them to their clients. A central coordinator shows the measuring devices and the values they provide at a time in a graphical interface that could, for example, resemble the building plan.

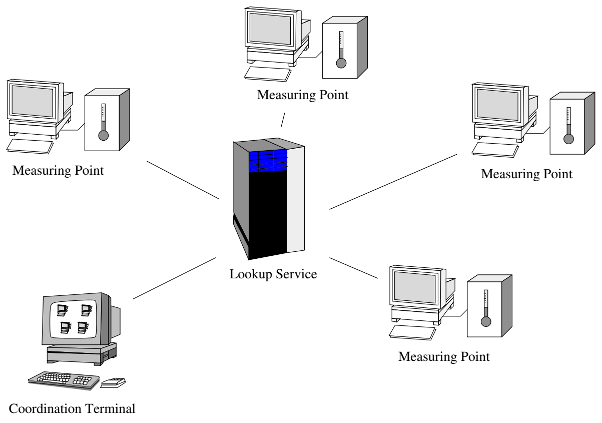

Figure 4. A JINI–based monitoring service for thermometer devices

The measuring devices currently are small Intel i386–based PCs with only minimum resources available. They contain a small A/D converter and an ethernet adapter, enabling them to pick up information from sensors attached to them and offer them through an IP socket. The measuring devices do not have the power to run a JAVA virtual machine. Provided with only a few megabytes of RAM, they rather run a native binary that reads out the A/D converter and implements a very simple protocol a coordinator can use to pick up the values. The coordinator again will use a proxy fetched from the Lookup Service to actually communicate with the measuring devices.

When switched on, a measuring device will try to locate a local JINI environment using IP multicast. If successful, the device will register itself with the Lookup Service using the procedure described above. Any coordination GUI will be informed by the Lookup Service that another measuring device has been installed and registered. The coordination GUI will fetch the device's proxy from the LUS, display another symbolized measuring point in the graphical interface and start collecting measuring information from the device, using the proxy it has previously downloaded. If the measuring device should eventually cease to answer requests from the coordinator or no longer renew its registration lease, the registration will be purged from the Lookup Service and the symbolized representation within the GUI will be removed.

5 Summary and outlook

The JINI architecture as introduced by SUN Microsystems provides very powerful means for the construction of distributed system architectures. Still, the JINI architecture bears a certain disadvantage in that it is completely based on the JAVA programming language, thus locking out all devices that are not able to provide the power and resources required to host a virtual machine and the JINI application itself. Even though SUN proposed a few approaches on how to integrate non–JAVA devices into a JINI environment, they all show the drawback that they rely on virtual machines provided by other participants of the environment. JINI consequently lacks a way to integrate non–JAVA participants without dependencies to third party.

This article introduced one possible strategy to achieve this goal. We showed, that JINI's current Lookup Service implementation can be modified to provide a special discovery/join protocol variant. In addition to the basic discovery and join protocols, the JINI environment is heavily based on a resource leasing mechanism. This article introduced a simple, message based protocol that gives non–JAVA parties the means to follow a basic JINI leasing strategy as fully JAVA capable would. Again, the modified leasing protocol can easily be provided by small, restricted devices.

Due to the problems one faces when integrating non–JAVA devices into a JINI environment, not all parts of the JINI architecture are good candidates for being directly accessible by non–JAVA parties. Still, we think there is one important aspect of JINI that will have to be considered to provide optimum integration, namely JINI service attributes. Within the JINI architecture, service attributes help to better describe service offers stored by the Lookup Service. Service attributes thus help to optimize service selection as offered by LUS. Consequently, any integration of non–JAVA parties into the JINI architecture will clearly gain from regarding and keeping the flexibility JINI offers with respect to dynamic service attributes. We are currently investigating on how the handle attribute changes on both the service provider's and the Lookup Service's side. Especially dynamic attributes demand a protocol to be followed when evaluating their value at runtime. We think that here again the use of a socket based protocol will suffice to find a good compromise between elegance with its integration into the JAVA philosophy and implementation complexity on the non–JAVA side.

References

[1] P. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, Februar 1996.

[2] W. K. Edwards. *Core JINI*. Prentice Hallo PTR, 1999.

[3] S. Microsystems. Jini architecture specification. Technical report, SUN Microsystems, Palo Alto, 1999.

[4] S. Microsystems. Jini device architecture specification. Technical report, SUN Microsystems, Palo Alto, 1999.

[5] S. Müller, K. Müller-Jones, and W. Lamersdorf. Global Trader Cooperation in Open Service Markets. In O. Spaniol, C. Linnhoff-Popien, and B. Meyer, editors, *Proc. Intl. Workshop on Trends in Distributed Systems*, volume 1161 of *Lecture Notes In Computer Science*, pages 214 – 228, Aachen, October 1996. Springer.

[6] K. Müller-Jones, M. Merz, and W. Lamersdorf. The TRADEr: Integrating Trading Into DCE. In K. Raymond and L. Armstrong, editors, *Proceedings of the 3rd IFIP TC6 Conference on Open Distributed Processing*, Proceedings of the ICODP, pages 476–487. Chapman & Hall, Februar 1995.