Abstract

The objective of this work is to reduce the code size and runtime of Jini and Java so the entire system software stack fits in the limited ROM/RAM space of a ubiquitous device. Having identified the size bottleneck being the use of RMI and unnecessary functionalities in the context of ubiquitous devices, we 1) propose to exchange service description instead of service proxy between the lookup server (LUS) and other Jini entities, which allows us to decouple the use of RMI easily, and 2) implement only the minimum service discovery functionality. This simplified form of Jini is referred to as Mini. We have achieved in implementing Mini with similar Jini API and in reducing the Mini/Java stack size to a mere 1133KB. That is approximately a factor of 20 smaller than Sun Microsystem’s implementation of Jini/Java and the size of Mini itself is approximately a factor of 4 smaller than the smallest Jini implementation we have come to know.

1 Introduction

Ubiquitous devices, for example mini-laptops, PDAs, mobile phones, and home appliances, have the potential of accomplishing collaborative tasks and bringing convenience to life. The challenge of realizing such a network of collaborative devices lies in how we handle the inherited heterogeneity, mobility, and limited resource problems. These devices can have very distinct hardware and software profiles, thus heterogeneous. They can move very dynamically among several networks, thus mobile. They can have very limited computation capability and mem-
ory space, thus resource constrained. System software for these devices must be able to deal with these problems.

Java and Jini are among the most prominent solutions to overcome the heterogeneity problem and allow service discovery for dynamic devices. This service discovery functionality is particularly important in this ubiquitous network context where the devices are migrating and services are often specific to the local environment. It is difficult to anticipate the kind of services that will be needed. Besides, these devices are limited in memory space. It is not practical to install all probable services all at once. We think, in this context, a minimum system software stack that supports dynamic service discovery and loading is much more appropriate.

Taking the hands-on approach, we have selected Java and Jini as the reference system. We have shown [5] that existing Java and Jini implementations, although supporting cross-platform and seamless service discovery, are too large in code size for the devices under consideration here. Following up that finding, we set out in this work with the objective of reducing the size of Jini and Java so the stack fits in the limited flash memory space. We think reducing system software size is critical to the success of a true ubiquitous network, i.e., devices everywhere. The reason is that the smaller the system software stack, the lower the manufacturing cost and the wider range of devices to be included in the collaborating mass – therefore increases the coverage and the impact of the ubiquitous devices.

Having identified the size bottleneck of Java/Jini stack being the use of RMI and unnecessary functionalities in the context of ubiquitous devices, we 1) propose to exchange service description instead of service proxy between the lookup server (LUS) and other Jini entities, which allows us to decouple the use of RMI easily, and 2) implement only the minimum service discovery functionality. This simplified form of Jini is referred to as Mini.

Using Compaq iPAQ as our experimental platform, we have achieved in implementing Mini with similar Jini API and reduced the Mini/Java stack size to a mere 1133KB. That is approximately a factor of 20 smaller than Sun Microsystem’s implementation of Jini/Java and the size of Mini itself is approximately a factor of 4 smaller than the smallest Jini implementation we have come to know. We next briefly compare our results to other related work and describe the critical components in Jini. Then we proceed to introduce the Mini architecture and present the differences between Mini and Jini quantitatively and qualitatively.

2 Related Work

There are a number of effort from the research and the commercial communities to extend the use of Jini to ubiquitous (or resource limited) devices. Sun Microsys-
tems introduced the *Surrogate Architecture* [7] in order to integrate limited devices into a Jini network. In the Surrogate Architecture, *surrogate hosts* act as Jini proxies for the limited devices. These limited devices can be attached to the proxy by any proprietary protocol. Although, the Surrogate Architecture enables limited devices to be part of the Jini concept, it radically reduces the degree of spontaneity offered by Jini.

Former investigations on the topic [5] showed that RMI is the bottleneck of the Jini stack. Attempts to improve the size and performance of RMI were realized by [14]. Others [3, 8], tried to avoid the RMI layer for Jini. They re-implemented Jini with a substitute for RMI. In [3], the RMI layer is substituted by a Remote Procedure Call (RPC) mechanism. They managed an implementation based on the Connected Limited Device Configuration (CLDC) Java platform. In [8], *NetObjects* are proposed to replace RMI. In the concept of *NetObjects*, proxy-objects are serialized at runtime and use reflection after deserialization on the remote Virtual Machine to discover the service methods. This method makes use of *server stubs* and HTTP servers unnecessary. However, *NetObjects* require classes from the Java 2 platform not available on small Virtual Machines. These solutions focus on alternative communication models to replace RMI. Our solution takes another step further. In addition to removing Jini’s dependency on RMI, we strip Jini down to include only the necessary service discovery functionality.

In [10, 13] they optimized the JAVA 2-RMI stack. *SavaJe*[10] is a Java Operating System (OS) for embedded devices. It provides the full Java 2 API including OS in only 12 MB ROM space. [13] is a hardware solution supporting a subset of the JAVA 1.1 platform in a small footprint of ~ 50 K. However, serialization and reflection is not supported. [9] developed a proprietary Jini implementation with a footprint of ~ 100 K running on the *TINI* platform [13]. The basic idea of this implementation is to shift the intensive processing of RMI to the client. This solution assumes that service providers have resource constraints, whereas service consumers do not. A service provider with limited resource runs its own lookup service in addition to its service locally. The *ServiceRegistrar* objects are modified to avoid the use of RMI. These solutions are proprietary and some hardware specific. Our solution with comparable code size is open source, software based, and inter-operable with the above solutions.

3 Approach

3.1 Jini

The Jini technology is a service oriented approach to realize spontaneous networking. Jini addresses the challenges of spontaneous distributed systems such as
robustness, self-healing, administration-freeness, and heterogeneity with five key concepts:

**Lookup.** The heart of Jini is the Lookup Service (LUS). The LUS acts as an extended registry for a Jini community. It is comparable to a naming service in distributed systems. Divers LUS may coexist and register mutually forming extended LUS structures. Both, service user (SU) and service provider (SP) use the discovery protocols to retrieve a `ServiceRegistrar` object from a LUS. This object is a RMI back-end object used to communicate with the LUS.

**Discovery and Join.** SU and SP find a LUS through the Jini discovery protocols. Two distinct discovery methods are available. Multicast discovery is used to discover unknown LUS. If multicast is not supported by the network or a well-known LUS is used, unicast discovery is an alternative. Unicast discovery requires the discoverer to know the address of the LUS. The result of both discovery processes is a `ServiceRegistrar` proxy. This proxy acts as a back-end object allowing a connection oriented communication with the LUS. Once the `ServiceRegistrar` proxy is downloaded by a SP, the SP registers its service by invoking the method `ServiceRegistrar.register()` on this object. Services are encouraged to use persistent, globally unique IDs providing unique identities in the system. The discovery and join procedure between a SU, a SP, and a LUS is pictured in Figure 1.

**Leasing.** In Jini, service or event registrations must be leased. A lease is a time-restricted interest that must be renewed periodically in order to remain valid. Lease times are determined by the lease grantor. Unrenewed leases are interpreted as expired registrations and must be handled consequentially. Leases provide a consistent picture of the system components when entities join and leave unpredictably.

**Events.** Jini provides mechanisms to send asynchronous events between remote processes. Event listeners must first register their interest. Therefore, event listeners must lease their interest for event reception.

**Transactions.** Distributed transactions allow to perform several distributed operations as if it was an atomic operation. This ensures that all operations are processed together or none at all. In a succeed transaction, all operations appear to occur simultaneously.

### 3.2 Reduced Functionality and Implementation Requirement

Earlier research [5] to run Jini on limited devices showed that RMI is a performance and memory bottleneck in the Jini stack. Furthermore, Jini uses the Java 1.2 RMI interface which is not compatible with the reduced Java platforms as `PersonalJava`, `EmbeddedJava`, or even unpresent as the case of Java 2 Micro Edition (J2ME). Re-
mote object activation, object marshalling, and the Java 2 security scheme have been introduced first since Java 1.2. Class reflection and the class loader mechanisms have undergone changes from Java 1.1 (PersonalJava and EmbeddedJava).

Instead of adding these lacking mechanisms, providing a slightly better performance and system software size, Jini was re-implemented from scratch. This procedure is motivated from the fact that computational resources in ubiquitous environments are too expensive or just not abound. Our approach focuses on the requirements of ubiquitous device computing while avoiding RMI for remote invocation of distributed objects, thus, reducing the computational and memory resource requirements of the Jini stack.

The primary goal was to develop a testbed for ubiquitous and spontaneous networking for interacting limited devices. The implementation has been called Mini because of its mini-mal resource requirements. The idea was to keep the basic key features of Jini in Mini. The following required features were identified:

- Any client or service must be able to discover a lookup service.
- Any service must be able to register with a lookup service.
• Clients and services must be able to renew registrations via leases.

• Any client must be able to request a service for usage.

• The lookup services have to return a list of registered services upon demand.

• The lookup services have to notify all entities of expired services.

All these features had to be provided by Mini. Furthermore, the following requirements regarding the wanted implementation were specified.

• Mini has to run in heterogeneous environments and on resource limited devices. A full Java 2 Virtual Machine is not required by Mini but instead a Virtual Machine conform to JDK1.1.8 as EmbeddedJava or PersonalJava.

• Mini has to be implemented without the RMI classes contained in the java.rmi package.

• Mini should consume as little resources (memory and CPU) as possible.

• In a first step, compatibility issues with Jini should not be of focus but only on resource consumptions and efficiency of the implementation. The Mini API should be similar to Jini and accessed intuitively for Jini programmers.

3.3 Mini

Mini is depicted in Figure 2 with the interaction of a LUS, a SP, and a SU. The required steps are explained in order for a SU to discover a SP via a LUS. Minor differences with the identical scenario in Jini (Figure 1) can be observed. Similar to Jini, the SP first makes a multicast or unicast discovery request to discover a LUS. After reception of a RegistrarProxy, the SP registers its service description, ServiceDescription via the register() method of the proxy. The essential difference with Jini is the ServiceDescription. It only contains a description of the service, such as the name, the location, or the service type, instead of a service proxy object. In Jini, this proxy object is later downloaded by the SU for invocation of the service.

At the same time a SU makes a discovery request and downloads the RegistrarProxy of the discovered LUS. The SU invokes the lookup() method, specifying its request with a service template, ServiceTemplate. The LUS replies with a list of registered service descriptions. Now it is the task of the SU to locate the ServiceProxy and download it from the SP. This proxy object is then used to invoke the actual service.
In the scenario depicted in Figure 2, a SP requests a service from the LUS. At the time of the request, the service has not yet registered. The SU later performs the lookup a second time. This time, the requested service is registered at the LUS and the LUS responds to the lookup request with the service description of the SP.

After this, the SU requests the service proxy class from the HTTP server, setup by the SP, and the SU then invokes the service through the downloaded proxy.

Figure 2: Mini - Interaction between LUS, SP and SU

Mini provides its own class loader (NetworkClassLoader). Classes, such as service proxies, can be downloaded dynamically from a HTTP server over the network. The service has to set up a HTTP server to enable this dynamic class loading.
The downloaded proxy object is used to communicate with the LUS. For this purpose, objects are serialized on the LUS and deserialized by the receiver. Unknown classes are downloaded dynamically by the class loader from a HTTP server. Reflection is used to find out the methods of the now downloaded class. This is also the way how a SU and SP interact, as depicted in Figure 2. In the case of Mini, objects are passed by value. Jini uses RMI for this object passing. In RMI objects are passed by reference. Proper handling of the SU and SP must be realized to allow objects to be passed by reference. The LUS makes use of events in order to advertise listeners when the object status changes.

4 Evaluation

4.1 Quantitative Comparison

The measurements in this section were done on three different platforms. A Compaq iPAQ H3760 [1] running Familiar Linux v0.5 [6] and Kaffe [4] CVS snapshot (19.02.2002) was used. A second iPAQ, model H3870 [2], running Microsoft PocketPC 2002 and the Jeode VM [12] was also part of the testbed as well as a desktop Pentium PC with a 870 MHz CPU running Windows NT v4.0 and Sun’s JDK1.1.3v01. These three platforms were used to measure the ROM and RAM consumptions of Mini and to illustrate the achieved improvements.

4.1.1 ROM Usage

The total size of the Mini classes, contained in Mini.jar, is 26 KB. This size is comparable to the core classes of Jini contained in jini-core.jar and reggie.jar. These two packages together require 258 KB. This is a reduction to the factor of ten. Obviously, this size improvement also implicates lacking features in Mini. These lacking features are highlighted in the next section.

The Mini package is implemented for use with PersonalJava or EmbeddedJava. Table 1 and figure 3 show the difference of Mini when used with distinct Virtual Machines. Sun’s Jini (v 1.1) implementation running on JDK 1.3.1 and RMI was used as reference. The Java 2 core classes consume 17 MB of ROM space (this value may vary for different Java 2 environments). With the size of the Virtual Machine, this configuration requires a total amount of 17758 KB. In contrast, Mini running on Kaffe consumes a total memory of 1133 KB as showed in table 3. Kaffe even includes classes unnecessary to Mini and could further be reduced as proposed in [5]. Mini running on Jeode also significantly reduces the size to 2926 KB. Again, the core classes of Jeode can be minimized with the use of a separate tool available by Insignia Solutions [12] down to about 500 KB. Mini on JDK
1.3.1 does not significantly reduce the memory footprint. This is due to the unused classes of the Java 2 core package.

Figure 3: Memory Footprint of ROM

<table>
<thead>
<tr>
<th></th>
<th>Jini on Java2</th>
<th>Mini on Java2</th>
<th>Mini on Jeode</th>
<th>Mini on Kaffe</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>500</td>
<td>500</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>core classes</td>
<td>17000</td>
<td>17000</td>
<td>2700</td>
<td>1007</td>
</tr>
<tr>
<td>communication package</td>
<td>258</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>TOTAL</td>
<td>17758</td>
<td>17526</td>
<td>2926</td>
<td>1133</td>
</tr>
</tbody>
</table>

Table 1: Memory Footprint of ROM
4.1.2 RAM Usage

In this section, the RAM consumption of Mini is deduced. Further tests are provided to also demonstrate the memory consumptions of Sun’s VM, Kaffe, and Jeode during execution of the LUS.

JProbe [11] was used to determine the heap size. The first measurement was realized on the desktop machine with Jini. Figure 4 shows the heap size and RAM usage during the registration of a Jini LUS with the RMI daemon (rmid). The first 15 seconds, the time it takes to register with rmid, are pictured in this figure. The minimum heap size required for this process is 1 MB.

![Figure 4: Heap Summary Chart for the Jini LUS](image)

In the second measurement the Mini LUS was executed on the same desktop machine. Figure 5 shows the heap size during the running process of the Mini LUS. During this time, no services were registered with the LUS. The only action performed by the LUS was sending update events to other LUS. For this measurement the event repetition time was decreased to 1 second. The heap size oscillates between 250 KB and 900 KB. The saw tooth form in figure 5 is due to the garbage collector.

![Figure 5: Heap Summary Chart for Mini LUS](image)

In a next step, different Java Virtual Machines were considered to compare the memory consumptions of Mini. These measurement results are summarized in Figure 6. These measurements were based on monitoring the task manager of the operating system. Then, the collected values were subtracted by the measurements of the LUS from previous step and plotted in Figure 6. The Mini LUS,
CreateLookup, was executed on the three platforms, introduced at the beginning of the section.

The first measurement was made on the desktop machine. At startup, a Mini LUS was launched. This process required 5300 KB of memory. This value linearly increased until it reached 5800 KB. This result correlates with the previous results made with JProbe (see Figure 5), where the heap size increased from 440 KB to 905 KB. We conclude that the Java 2 runtime environment is using 4900 KB of memory during runtime in this specific configuration.

The second measurement was completed identically, but on the iPAQ H3870. Again, by examining the task manager, the required memory fluctuated between 3000 KB and 3500 KB. Once more, an increase of 500 KB was observed before the memory was freed by the garbage collector. The Jeode VM used approximately 2600 KB of memory\(^1\).

\(^1\)The Jeode VM could eventually be tuned for this purpose according to [12]. No tuning was done

---

![Memory usage](image.png)

Figure 6: Memory Consumption of Kaffe, Jeode, and JDK 1.3.1 with and without LUS
The last measurement was realized on the iPAQ H3760. The Kaffe VM required around 800 KB of memory and consumed with the LUS approximately 1700 KB before the garbage collector became active.

These results show that approximately 80 percent of total memory can be saved when running the Mini LUS on the Kaffe VM compared to the Sun’s VM.

4.2 Qualitative Comparison

Various differences between Mini and Jini are caused by the optimization of the system software size. Among the most important distinctions in functionality are:

• Mini lookup servers only store ServiceDescription objects of registered services unlike proxy objects in Jini. SPs do not upload their proxy classes or objects to lookup servers but make them available for download at an HTTP server for SUs. This way, service registration entries require less memory at LUS and the resource consumptions are better distributed among the entities.

• Mini lookup servers do not have to register objects with the remote method invocation daemon (rmid). The Mini lookup server is a stand-alone application. On the other hand, remote activation and deactivation of the lookup server is not possible with Mini.

• In mini, the service interface must not be present on the client. There is no need for well-known interfaces. Nevertheless, service objects can be downloaded dynamically by clients. The client uses class reflection to determine the methods provided by the service.

• Leases are handled differently in the Mini implementation. Lease times are specified by the requester in contrast to the lease grantor in Jini. In addition, lookup services do not remove service registrations of unused services unless the service did not renew its lease.

• Mini does not identify services with unique IDs. Services are uniquely identified with ServiceDescription containing a name associated with a network address. Therefore, the challenging task of generating globally unique IDs is avoided.

5 Summary and Outlook

Mini is a minimal platform for collaborating, heterogeneous devices. It provides the minimal features for robust and dynamic service discovery in the context of
ubiquitous computing. With Mini, we achieve in reducing the size of Jini by a factor of 10 and the size of Java/Jini stack by a factor of 20 (See Table 1). The former size reduction is primarily contributed by implementing only the necessary service discovery functionality. Replacing service proxies with service descriptions also helps in reducing the size of Jini, but the major effect of this change is the decoupling of RMI for which we are able to do away without a full-blown Java virtual machine. That is the reason that we are able to use Mini with a variety of Java virtual machines without specific RMI support. This further contributes to the factor of 20 size reduction for the combined Java/Mini stack. The stack is now 1133KB large in size and fits in all systems with 2MB or larger ROM space which covers all the PDAs and most embedded system development boards we come to know.

There are two potential directions of future work. One is the interoperability issue between Mini and Jini that we inherit by changing the communication model. We propose a translator gateway to facilitate such inter-operation. The mechanism is elaborated in section 5.1 below. Furthermore, Jini, and Mini so far, assumes at least one lookup server per subnet and discovers services within the subnet. In a multi-hop device network consisting of several subnets, the lookup servers will not be exchanging their service registries across subnets so devices on one subnet will not have a chance of discovering services provided elsewhere. To overcome this deficiency, we currently adopt a solution similar to Jini’s where all lookup servers within a community are known a priori so local service registries and event notifications can be exchanged among all lookup servers on different subnets. Opting for an adaptive solution, we propose, in Section 5.2, a unicast-routing-inspired solution that assumes no a priori information.

5.1 The mini-Jini compatibility problem

The idea of connecting the two worlds is to place a device in between that acts like a translation gateway. Through this gateway, mini lookup servers communicate with Jini lookup servers and vice versa. Clients however, only talk to lookup servers from their own world. The goal is that services from both world can have entries in any lookup server. An illustration of a gateway between the two worlds is given in figure 7.

Service users from the Jini world should be able to use all services, including the Mini services. All the gateway needs to do is to convert and register Mini lookup service as a service to the Jini community. The Jini service users can then discover the Jini lookup service and interact to obtain services provided in the Mini community.

For the other direction, service users in the Mini world should at least be able to obtain non-RMI-based services. The role of gateway here is to translate
ServiceProxy to Mini-compatible ServiceDescription objects. From these ServiceDescription objects, they can learn where to load the corresponding proxy objects. Due to the fact that RMI is not supported at all for the Mini service users, Jini services that make use of RMI will never work for Mini service users. Interoperability in this direction is not completely warranted, thus as indicated by the dashed line at the bottom of figure 7. This gateway-based approach for inter-Jini-Mini-operability is currently undergoing investigation.

5.2 Subnet

As already depicted in Section 3.3, Mini relies on subnet broadcast for discovery and multicast for event notification. This subnet broadcast mechanism does not forward broadcast traffic further to other subnets. All packets sent through broadcast are received and computed only by network interfaces within the same subnet. Also, it is not straightforward to inject broadcasts into other subnets by sending packets to the broadcast address of another subnet, which might be considered a security thread if the intension of the traffic is not clarified.

To include clients connected to a different subnet in the Mini community, a client has the possibility to start a unicast discovery process if the address of the lookup server is known. With the use of the unicast discovery, a client can load the RegistrarProxy object of a specific lookup server, not just off of a random one.

Alternatively, clients can discover through multicast to communicate with a group of lookup servers. In contrast to broadcast packets, multicast packets will
only be received by hosts that are interested in receiving them. By joining a multi-
cast group, the network interface knows that this process wants to receive all data
that is send to this group. If no process on a host is a member of a multicast group,
its network interface does not have to forward any packets with a destination ad-
dress other than its unicast address to any process. This reduces traffic compared
to broadcasting. The problem here is, however, multicast routing might not be
deployed over all the connecting subnets of a Mini community.

In comparison, Jini’s solution to join different subnets is the concept of feder-
ated lookup services. Every subnet has a lookup service running that is talking and
updating others over a well known connection. This makes the discovery process
transparent to any subnet and all clients in the network area can join the Jini fed-
eration. Jini solves this problem with the implementations in the fiddler.jar
package.

A similar approach has been implemented in Mini. Every lookup server keeps
a local list of other known lookup servers. This list has to be updated by users with
the IP addresses of all known lookup servers running on different subnets. This re-
quires a priori information about the lookup servers, most probably pre-configured
by the system administrators. The lookup server then forwards the events it re-
ceives via multicast from its own subnet to all the lookup servers in its list using
unicast. A lookup server receiving such a unicast event notification, on the other
hand, then multicasts it to the Mini community in its own subnet. Events received
over unicast are never forwarded to other lookup servers. Infinite loops of multicast
event notifications are avoided this way. However, this method implies that every
lookup server has to know all other lookup servers to guarantee that all events of
all mini members are received by all members in every subnet.

Identifying this being a similar problem to unicast routing [15], we are look-
ing into a hop-by-hop relaying approach to locate all lookup servers within a Mini
community. In that, we forward the service registry (subnet addresses) per lookup
server (responsible router) to neighboring subnets. The lookup server on a neigh-
boring subnet will aggregate information coming from several neighboring lookup
servers and continue to relay the aggregated service registry (i.e. updated subnet
address and responsible route pairs) to all the neighboring subnets except the one
from which the lookup server receives the initial service registry.

We think this is another area of potential which overlaps with the problems
studied by the ad hoc network community [16]. In that, several routing mech-
anisms have been proposed to provide routing information for a mobile ad hoc
network. The problem we are facing is exactly to provide routing information per
service for a mobile ubiquitous device network. We are surveying at the moment
for efficient, low-cost ad hoc routing solutions that are applicable in this domain.
References


