HYBRID JINI FOR LIMITED DEVICES

VINCENT LENDERS, POLLY HUANG AND MEN MUHEIM
ETH Zürich
E-mail: {lenders, huang}@tik.ee.ethz.ch, men@ife.ee.ethz.ch

We envision a future of heterogeneous mobile devices collaborating spontaneously and bringing convenience to life. Taking a practical approach, we study the feasibility of integrating limited devices into the Jini, Java, and Linux paradigm. With careful evaluation and system hacking, we manage to fit the software stack, excluding Java's Remote Method Invocation (RMI) support, in a small 1.85-Mbyte space. However, we also identify that including RMI support will exhaust all the remaining space. We propose, as a short-term solution, to re-implement Jini with light-weight communication alternatives and to use a hybrid lookup server to inter-operate RMI- and non-RMI-supported devices. For the long term, we advocate minimalism and call for community-wide standardization effort to the system software development.

1 Introduction

We envision a future that computers and electronics will be mobile and collaborate spontaneously, bringing convenience to life. Two immediate problems before realizing such a future are ways to handle heterogeneity and mobility of these devices. These devices, for example mini-laptops, mobile phones, and home appliances have very distinct hardware and software profiles. The task of configuring and inter-operating them will be too difficult for average users, not to mention that these devices are likely to be mobile and whenever they enter a new environment they need to be re-configured.

To counter the problems of automatic configuration and seamless inter-operation, the industry and research community have offered a few solutions [1,2] that encompass (1) a resource discovery/management protocol and (2) a framework of uniform programming interface and code base. One prominent example is the Jini [3] and Java [4] combination. In the Jini/Java paradigm, devices can come and discover (or be discovered by) resource in the network automatically. While responding, the devices register proxies allowing other members in the network to operate them.

As a feasibility study, we are building such a Jini/Java-based spontaneous network with various types of devices, mainly full-fledged computers, mobile PCs, and limited electronics. Connecting limited electronics is the most interesting and challenging case. By limited electronics, we mean personal, home, or office electronics. They are essentially micro-computers with compact system architectures and limited flash memory, and sometimes also referred to as *embedded systems*.

Taking a hands-on approach, we set ourselves off to install the Jini/Java stack on a development board for embedded systems. This turns out to be a non-trivial

task and gives rise to a serious system software problem – its size. It is profoundly difficult just to fit a native Jini/Java/Linux stack onto a typical embedded system board, not to mention extra applications required to run on top of the system stack.

After intensive system hacking and careful evaluation, we find bottleneck of the size problem being Java's Remote Method Invocation (RMI) support. It is humongous, relatively to other parts, and current implementation of Jini depends on it. While RMI offers elaborated remote execution support, it is not really necessary for communication in Jini which can be rather primitive.

We propose, as a solution to scale system software size, a light-weight Jini implementation based on IP socket or XML. That is to implement Jini without using any RMI interfaces and avoid including RMI support in the software stack at all. To warrant backward compatibility with existing implementation, we take a hybrid approach to differentiate existing RMI-based and the light-weight implementations, thus enabling execution of non-RMI proxies on RMI-supported devices but not the other way around.

More importantly, we call for standardization activities towards defining minimum system software stack for mobile devices. In the meantime, we urge developers to refrain from RMI and the community to re-examine its scalability.

In short, our *contribution* includes a) a successful port of system kernel and free-license VM for limited devices, b) a qualitative and quantitative evaluation of a minimum software stack, c) a 35% reduction of the Jini/Java/Linux stack without RMI support to *1.85 Mbytes*, and d) a light-weight hybrid Jini solution to overcome the system software size problem.

2 Approach

Jini devices require hardware platforms with network connectivity and the ability to execute Java code. Two different minimal hardware approaches are possible. The first solution is an embedded device with network connectivity that runs a Java Virtual Machine (VM) on top of any kind of operating system. Typical embedded devices of this kind are the Developer Board LX from Axis [5] and the NetSC520 Demonstration Platform from AMD [6]. Both platforms run embedded Linux and communicate via Ethernet. The second solution is to use embedded boards with Java processors. Unlike traditional microprocessors, which must convert Java byte-code into the processor's native language, these processors can operate directly from Java byte-code. aJile Systems [7] provides a development board with their Java processor called aJile. Another solution is provided by Systronix [8]. They developed a hardware Java platform with their Tiny Network Module (TINI).

We decided to use the Developer Board LX from Axis for our further implementations. This board provides the best scalability regarding resources, size, and price.

It has the following features: 100MIPS 32 bit CPU, Ethernet 10/100Mbps, 2 RS-232 serial ports, 2 parallel ports, 2 Mbyte FLASH and 8 Mbyte DRAM. The FLASH size can be updated to 4 Mbytes.

Operating Jini on this platform with restricted resources requires strict size limitations regarding the operating system and the VM. Linux suits well as target operating system for our device. The two main reasons are the compact size of Linux and its open license policy. We use a Linux port as operating system.

Different Java versions from different parties are candidates to be ported to the developer board. Sun Microsystems introduces the Java 2 Micro Edition (J2ME) [9] as Java platform for limited devices. J2ME is divided in two complementary configurations: the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). Both configurations introduce a new VM. The CDC contains the C virtual machine (CVM) and the CLDC contains the K virtual machine (KVM). CLDC would probably better suit for the developer board.

Kaffe [10] is an open implementation of Java for embedded and desktop systems. It is an implementation of the PersonalJava 3.0 specification and requires no source code licenses from Sun Microsystems. The major advantages of Kaffe are: compact size, easiness of porting to new platforms (operating systems and architecture), and it is open source software. For these reasons, we decided to use Kaffe instead of J2ME for our implementation.

3 Evaluation

The main task of this work is to fit the required software on the 4-Mbyte FLASH ROM. The required software includes the operating system, VM, Jini core classes and Jini service classes.

Embedded Linux. The CPU on the developer board is the ETRAX 100 LX. The source code of Linux 2.4 includes the CRIS architecture for ETRAX 100 LX. Axis provides a small distribution of Linux for their developer board. We have optimized this distribution and reduced the size of the operating system (kernel and basic utilities) to *800 Kbytes* (55 % of the original size). Breakdown of component sizes in the FLASH ROM is depicted in Figure 1.

Kaffe VM. As described in the last section, we use Kaffe as Java VM. Kaffe has not been ported to the CRIS architecture yet and it is part of our contribution to port Kaffe to this new architecture. The classes in Kaffe are bundled in eight jar files. The core Java classes are part of the Klasses.jar file. The original classes are shown in Table 1.

For memory saving reasons, we port only a subset of the core Java classes. The awt, applet, beans, math, and sql packages are removed since they are not useful for the given board as a Jini device. Table 1 shows the size of the different packages and

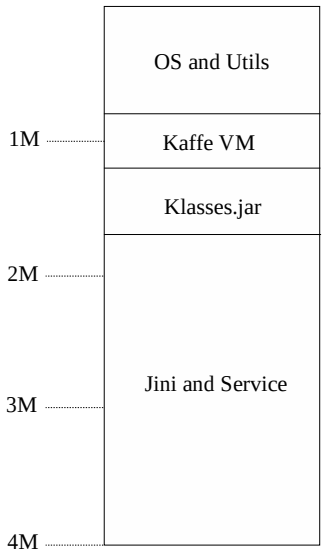

Figure 1. Flash Reservation (4 Mbytes)

the removed packages. The original size of the Java core classes in Kaffe was 886 Kbytes. We could reduce the size of the core classes to *525 Kbytes*.

The VM was compiled with the gcc-cris cross-compiler. gcc-cris is a port of the GCC[11] to the CRIS architecture. We had to compile with static libraries since there is no support for shared libraries at the current time. The executable Kaffe VM for the CRIS architecture takes about *400 Kbytes* of memory space. Figure 1 shows the size of the Kaffe VM and the Java classes in respect to the whole FLASH ROM.

Jini. Jini is just a set of additional Java classes. The basic Jini class files are packed in three jar files: jini-core.jar, jini-ext.jar, and sun-util.jar. The Jini core classes are part of jini-core.jar and jini-ext.jar. The classes in sun-util.jar are not part of the official Jini distribution. These classes are helper classes from Sun to help developers to write their services and clients.

The major problem we encountered with the Jini core classes is that they require Remote Method Invocation (RMI). Our light-weight Java port does not support RMI.

package	**package**	**size [K]**	**jar-compressed size [K]**	**removed**
java				
	lang	428	97	
	awt	944	213	r
	util	588	133	
	net	168	38	
	security	184	42	
	io	324	73	
	text	112	25	
	applet	20	5	r
	beans	104	24	r
	math	20	5	r
	sql	80	18	r
kaffe				
	lang	76	17	
	awt	60	14	r
	util	100	23	
	net	104	24	
	security	40	9	
	io	420	95	
	text	40	9	
	applet	36	8	r
	beans	40	9	r
	jar	12	3	r
	management	12	3	
Total		3920	886	525

Table 1. Package Size and Reconfiguration of Kaffe

RMI requires too many computational resources to run on the board. In Table 2 and 3, the core packages of Jini are listed with their RMI dependence.

Our solution is to remove the RMI dependence from the Jini classes and to replace RMI's functionality with other mechanisms that scale for embedded devices and our Kaffe VM. Regarding the size restrictions, we have to fit the light-weight Jini and service classes in around *2.3 Mbytes* of memory space. Next we discuss the solution in detail.

Package	Function	RMI
net.jini.core.discovery	support for unicast discovery	yes
net.jini.core.entry	definition of Entry objects for type definition	no
net.jini.core.event	support of remote events	yes
net.jini.core.lease	support classes to implement leasing	yes
net.jini.core.lookup	interface and support classes to communicate to the lookup service	yes
net.jini.core.transaction	support classes for clients of the transaction manager	yes
net.jini.core.transaction.server	support classes for services that participate in transactions	yes

Table 2. The Jini Technology Core Platform (JCP)

Package	Function	RMI
net.jini.admin	interface to provide common ways of administrations	yes
net.jini.discovery	support for unicast and multicast discovery	yes
net.jini.entry	utilities for Entry objects	no
net.jini.event	classes related to the event mailbox service	yes
net.jini.lease	renewal of leases	yes
net.jini.lookup	assisting classes for lookup service joining and communication	yes
net.jini.lookup.entry	attribute types that services can use for registration with the lookup service	no
net.jini.space	JavaSpace support	yes

Table 3. The Jini Technology Extended Platform (JXP)

4 Solution

To solve the size problem, we can, on one extreme, add more memory on the limited devices or on the other extreme take the surrogate approach [12] that suggests to leave the limited devices as they are and to enable their participation through a Jini-capable delegate. Adding memory results in higher manufacturing cost which is not desirable from the industry's point of view. The surrogate approach, although solves the size problem, requires configuration of the Jini delegate on limited devices, which brings

us back to the configuration problem that Jini is designed to eliminate at the first place; thus not desirable from the users' point of view.

We propose a *hybrid* solution that solves the size, cost, and configuration problems. Having identified the bottleneck of the system stack, we suggest to replace communication part of Jini implementation, currently done by heavy-weight Java RMI, with light-weight alternatives, such as IP socket or XML. In Table 2 and 3, we list the RMI-dependent subset that we intend to tackle. Changes to ensure minimum discovery and maintenance will influence both the limited devices and the lookup server. Thus, a hybrid lookup server, referred to as LUS+, needs to be in place to handle both native Jini and non-RMI Jini devices in the network. In case a non-RMI device joining a Jini network which is not capable of handling RMI-based and non-RMI-based discovery, one will have to fall back on the add-more-memory or surrogate approach.

There is yet a backward compatibility problem. In case of a non-RMI device joining a Jini network assuming RMI fully supported, a LUS+ may naively pass RMI-based proxies to the non-RMI device and expect them to work. The solution will be to filter out RMI-based ones at the lookup server so limited devices will not receive proxies that they could not handle. Note that filtering of non-RMI proxies is not necessary. Devices with RMI support will not have any problem executing the non-RMI proxies.

It is obvious that we cannot afford to communicate limited devices using space-hungry RMI. Although the proposed light-weight hybrid solution helps alleviate the problem, there remains several critical questions to address. For instance, *do we absolutely need RMI as a means of communication in the context of Jini* or *is RMI inevitably large in size?* For the moment, we would urge developers to refrain from RMI, but for the long term, the community needs to take a close look at the scalability of RMI and to work towards defining a minimum set of communication interfaces so that limited devices (an important element in the future mobile network) can move and collaborate effortlessly.

References

1. Microsoft, *Universal Plug and Play Forum*, http://www.upnp.org
2. Object Management Group, Inc., *CORBA Forum*, http://www.corba.org
3. *Community Resource for Jini Technology*, http://www.jini.org
4. Sun Microsystems, *Source for Java Technology*, http://java.sun.com
5. Axis Communications, *Developer Board LX*, http://www.axis.com/products/dev_board/index.htm.
6. AMD, *NetSC520 Demonstration Platform*, http://www.amd.com/products/epd/desiging/evalboards/21.netsc520e/index.html

7. aJile Products, *aJile*, http://www.ajile.com/products.htm
8. Systronix, *TINI*, http://www.systronix.com/home.htm
9. Sun Microsystems, *Java 2 Micro Edition*, http://www.java.sun.com/j2me/, 2001.
10. Transvirtual, *Kaffe*, http://www.kaffe.org, 2001.
11. GNU, *The GNU Compiler Collection (gcc)*, http://www.gnu.org/directory/gcc.html, 2001.
12. Jini Community, *Surrogate Project*, http://developer.jini.org/exchange/projects/surrogate/, 2001