Middleware-Integration of Small Devices

Michael Mock
Fraunhofer-Institute
Autonomous Intelligent Systems
53754 St. Augustin
michael.mock@ais.fraunhofer.de

Stefan Couturier
B-IT Bonn-Aachen International
Center for Information Technology
53754 St. Augustin
stefan.couturier@smail.inf.fh-brs.de

Abstract

The integration of small mobile systems and devices into existing computing infrastructures requires efforts on the programming and on the networking layer. Both layers are often addressed by middleware that offers simple abstractions at the programming layer while implementing them on the networking layer. This paper focuses on JavaSpaces and Jini, a middleware provided by Sun. JavaSpaces implement a shared tuple space, a popular abstraction at the programming layer. Jini supports JavaSpaces on the networking layer. Unfortunately, Jini requires a full Java Virtual Machine and does not run on small systems. This paper presents an alternative implementation of the JavaSpace abstractions that is independent of Jini. It allows integrating transparently small devices into existing infrastructures while using the usual JavaSpace abstractions. Design, implementation and results are presented.

1. Introduction

Small devices, like mobile phones or personal digital assistants, are becoming more and more popular. Also, equipment and devices in industrial automation is more and more equipped with (limited) computational power. In order to use these devices with existing computing infrastructure, common networking facilities and programming abstractions for distribution must be provided.

The image of a process in industrial automation can be regarded as an example of a widely used abstraction in distributed programming, namely hiding distribution by providing shared access to distributed data by reading and writing shared variables. This shared data space approach has its counterpart in distributed systems at various places: its is found in distributed shared virtual memory systems that share data at the page level, in agent-based and multi-robot systems that are based on blackboard architectures, and in coordination languages such as Linda [21], that allow sharing on the level of structured variables.

The JavaSpace middleware provided by Sun [17] implements the shared data space model for Java objects (called entries in the JavaSpace). It is part of the Jini frameworks for mobile networked devices [15]. Hence, the JavaSpace middleware a promising candidate for the integration of small devices in distributed processes.

However, Jini / JavaSpaces require a full Java Virtual Machine, which is not supported by small devices in many cases. The reason for this is that the implementation of Jini and JavaSpaces relies on another very popular abstraction for distributed programming, namely remote procedure calls which are intrinsic in Java as RMI (Remote Method Invocations) and object serialization. The support of these powerful general purpose distributed computing abstractions blows up the Java Virtual Machine. Hence, for supporting small devices, Sun introduced a limited Java configuration called MIDP (Mobile Device Information Profile) [20] with an own virtual machine, the K Virtual Machine. This virtual machine does not support RMI and object serialization. As a consequence, small devices using the MID profile cannot directly communicate with JavaSpaces because of missing RMI and serialization capabilities. So, the problem is to implement a middleware that provides JavaSpace communication for MIDP.

The solution presented in this paper relies on implementing the shared dataspace abstraction directly on the messaging layer, thus removing the need for a general purpose RMI and serialization support. A distributed client - proxy architecture, in which the client is completely based on MIDP and works on the small device, achieves this. The client implementation, denoted as Small JavaSpace Client SJC), supports the standard JavaSpace classes and interfaces. Hence the same API is available for MIDP JavaSpace client as for a “real” JavaSpace. The proxy is installed on a machine that is running a full Java Virtual Machine – mostly on the host that is running the JavaSpace itself – and converts the network messages to actual JavaSpace entries and operations.

All in all, this solution allows writing, reading and taking JavaSpace entries and basic lease handling for MIDP. Standard JavaSpace interfaces can be used on the MIDP device, so code has not to be rewritten. Furhtermore, hosts running a full Java Virtual Machine can communicate with the small device via JavaSpace with-

0-7803-9402-X/05/$20.00 © 2005 IEEE

out even being aware of the client – proxy solution. Performance tests show that this distributed architecture is even faster than a usual JavaSpaces implementation. As application example, the Small JavaSpace Client and proxy are integrated into a project working with small mobile robots.

Chapter 2 introduces the technological background of this work and discusses related work. Chapter 3 presents the client - proxy architecture for JavaSpace and some aspects. Performance measurements as well as an application example are found in chapter 4. Finally, some conclusions and references are given.

2. Background and related work

2.1. Java

Java and its runtime additions can be uses as a complete middleware framework for web programming, distributed enterprise applications, and embedded applications. The Java 2 Standard Edition (J2SE) offers an object oriented programming language, which can be used platform independently [2]. As Java programs are executed on a virtual machine, the Java Virtual Machine (JVM), they can be used on all platforms for which such a virtual machine exists [2]. Distributed computing with Java is based on Remote Method Invocation (RMI). It offers the possibility to remotely invoke methods of Java objects on other Java Virtual Machines. This other virtual machine does not need to be run on the same host [16].

2.2. JavaSpace / Jini

Jini technology was developed to set up flexible, easy administered networks supporting dynamic services and devices. These services are entities that can be added and deleted flexibly [15]. Communication between services is established by using RMI. JavaSpaces, a part of Jini, implements the tuple space abstraction known from Linda [21] and provide a shared virtual object pool, which e.g. can be used to store data. This data is organized in so called entries. An entry is an object consisting of fields, which are objects themselves. Entries can be looked up by templates [17]. There are four kinds of operations that can be executed on entries and templates:

1. You can write an entry into a JavaSpace (write)
2. You can read an entry, which matches a template, from a JavaSpace (read).
3. You can read and also remove this entry from the JavaSpace (take).
4. You can listen to events that are generated by writing entries into the JavaSpace (notify).

Every component that is connected to a JavaSpace can do all four operations on the entries in the JavaSpace. Multiple operations can be grouped as transactions [17].

An application writing an entry to the JavaSpace can limit this entry's lifetime. The write method returns a lease for the entry by which the lifetime can be modified [17].

2.3. MIDP

The most prominent configuration for Java on small devices is the Mobile Information Device Profile (MIDP) [20]. MIDP is a special profile of the Connected Limited Device Configuration (CLDC), a limited Java implementation and thus part of the Java 2 Micro Edition (J2ME). It is implemented in most mobile phones and available for nearly all kinds of PDAs. Nevertheless there also exist another profile and also another configuration, the Connected Device Configuration (CDC), which itself contains several profiles (see table 2.1). Although theses profiles may be applicable for many devices for which a KVM exists and CDC supports more functions needed for JavaSpaces, MIDP has become a quasi standard and is thus used in our work.

MIDP applications are run is a small virtual machine, the so called K(ilo) Virtual Machine (KVM). They are based on a special class which is called MIDlet [20]. This class handles the life cycle of the program. Life cycles can be subdivided into the states of the MIDlet. A MIDlet can be loaded, active, paused, or destroyed. For each of these states a corresponding function exists which is called by the program manager.

The name "Kilo" Virtual Machine comes from the attempt to reduce the virtual machine's size to about 100 kByte. For this, some functions of the standard Java Virtual Machine, which are needed for communicating with JavaSpaces, are excluded (e.g. RMI, serialization). Moreover the profile does not allow for security reasons dynamic class loading from sources different than its own JAR file.

As the KVM only needs few resources (a ROM of 136 kByte and a RAM of 32 kByte), it is ideal for the use with small, resource constrained devices.

Edition	Configuration	Profile	VM	ROM	RAM
J2ME	CLDC	MIDP	KVM	136K	32K
J2ME	CLDC	PDA Profile	KVM	512K	
J2ME	CDC	Foundation Profile	CVM	1.0M	128K
J2ME	CDC	Personal Profile	CVM	2.5M	1.0M
J2ME	CDC	RMI Profile	CVM	2.5M	1.0M
J2SE	—	—	JVM	98M	64M

Table 2.1: Overview virtual machines based on [14]

2.4. Related work

As discussed in sections 2.2 and 2.3, Jini is ideal for mobile devices and MIDP is ideal for small devices. The problem is that one cannot use Jini services like JavaSpaces from MIDP because of the missing RMI and serialization implementation. So integrating small devices into a Jini environment has received much attention in research and development.

In the following, standardization and research efforts fitting into this line of research presented. Surrogate [1] [3], the most prominent of them, is an architecture developed by Sun. It is based on the idea to have an gateway between a Jini network and a device which is not able to directly take part in this network. This gateway consists of a Surrogate Host running on a so called "Host-capable Machine" [1]. Moreover the Surrogate Host implements a connection to the external, not-Jiniable device, a so called Interconnect. Via this Interconnect the surrogate, which contains all necessary information about the small device, is sent to the host. As the Interconnect can be used with a broad range of connections (LAN, Bluetooth ...), it needs an own protocol. The JiniME [10] extends Surrogate to a complete framework for providing Jini capabilities to small devices. Integrating small devices in the surrogate approach requires adopting the application code to the Interconnect protocol. A similar conclustion holds for the solution presented in [13]. It implements the surrogate architecture on mobile phones based on their MIDP-capability while changing the MID profile Moreover a new surrogate protocol is suggested. So, all in all, three things are different to the solution presented in this paper: The specialization to cellular phones, the usage of surrogate- instead of JavaSpace commands and the changing of the MID Profile.

In [12], access to Jini-services with MIDP 1.0 is achieved by tunneling Jini commands through HTTP. The solution enables the usage of arbitrary Jini services, but not with the standard Jini commands.

The Open Services Gateway Initiative (OSGi) defines a complete computing environment for networked services based on Java. In this environment management systems exist, which can manage any devices [6]. The environment supports many standardized services like HTTP, XML, UPNP, XML-HTTP and Jini, but it also has its own API [5]. Thus it is usable for many operating systems and can communicate with a usual Jini environment.

The main aim of Many-to-Many-Invocation (M2MI) is to work on collaborative systems that use wireless networks as communication platform [7]. It is usable for both mobile and fixed devices. Moreover it is based on a newly developed protocol, the Many-to-Many-Protocol (M2MP). This protocol makes use of the network's inherent broadcast nature and neglects routing messages from device to device [7]. Furthermore RMI is implemented to M2MI by giving devices references to the invocated method, so called handles [8]. Based on this framework the Anhinga project was started to develop a complete infrastructure for collaborative applications.

Main differences of those open standards to the Small JavaSpace Client are the usage of own commands, but these commands enable access to arbitrary Jini services.

Inca X ME, JMatos and YoCOA are three commercial solutions for integrating MIDP into JavaSpaces:

Inca X offers an own gateway (X ME gateway) to make MIDP applications communicate with Jini Services. For this it offers three software editions, the Community, the Standard and the Enterprise Edition. The Enterprise Edition provides some additional administration [4].

In contrast to the Small JavaSpace Client, Inca X offers access to arbitrary Jini functions. This access is provided via own API commands.

PsiNaptic JMatos is based on the idea that Jini can be run without using RMI [9]. Moreover the LookupService has to be intrinsic to the device's purpose. If a lookup service resides on a device which only offers services, this device needs only few resources.

JMatos provides an own software and not an extension to MIDP. So it differs from the Small JavaSpace Client in the fact that it may not be usable with every device that is able to run a K Virtual Machine for MIDP.

YoCOA provides a limited RMI for CLDC and MIDP, so that any J2ME device can manipulate objects on the server [11]. Data transmission in YoCOA is based on HTTP. Thus it allows object transmission, but is not directly intended to enable communication with JavaSpaces for small devices.

3. A Small JavaSpace Client

3.1. Architecture

Any application based on JavaSpaces has a fixed communication layers structure shown in figure 3.1.

Application
JavaSpace
Jini
Remote Method Invocation (RMI)
Messages (TCP/IP)
WLAN

Figure 3.1: Communication layers for JavaSpaces

This figure shows that for the use of a sophisticated communication technique like JavaSpaces several layers are necessary in addition to standard TCP/IP communication. As discussed in the chapter 2.3, the MID profile

does not offer these layers for several reasons: Neither is an application allowed to load classes dynamically from outside its own JAR file. Nor can you execute any command on another host, because RMI is not included. Furthermore serialization is not implemented in MIDP.

So, for enabling a MIDP device to communicate with JavaSpaces, a new approach has to be found. Usually new approaches concerning this topic result in new, specialized commands used for writing, reading or taking entries to / from the JavaSpace. Using this approach requires learning how to use these new commands. As this is very time consuming, the task comes up to develop an architecture that offers the standard JavaSpace commands for MIDP without changing the profile.

For this, the JavaSpace classes and interfaces have to be rewritten for MIDP with the same API as on a full Java Virtual Machine. Moreover, the architecture has to use the communication facilities offered by MIDP for writing, reading and taking entries to / from the JavaSpace and lease operations. These facilities are restricted to Transport Control Protocol (TCP) and User Datagram Protocol (UDP) connections. In contrast to full Java, MIDP is not able to use UDP multicast. Hence for both protocols, the client must know the IP address of the Small JavaSpace Proxy. This moreover means that the main advantage of UDP against TCP is gone. So TCP is chosen, as it is more reliable than UDP.

Writing, reading and taking entries requires sending all necessary information for this via a TCP connection. So at least the command (e.g. “write”), the timeout information and the entry must be transmitted. Furthermore, leases or entries may be returned to the client.

As serialization is not implemented in MIDP, objects (like entries) cannot be sent directly via MIDP socket connections, but only streams of byte arrays. So all objects that have to be sent from a MIDP device to the JavaSpace must be converted to byte arrays. These arrays have to be reconverted to objects on the JavaSpace side and forwarded to the JavaSpace. This task is performed by a proxy. So, as a result, in order to use socket messages for enabling MIDP to communicate with a JavaSpace, a distributed architecture, consisting of a client (on a small device) and a proxy implementation, is developed:

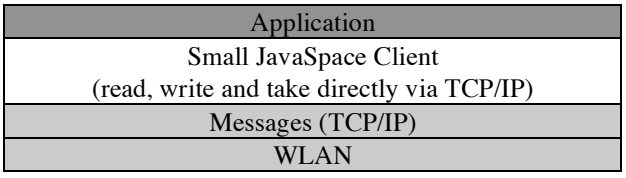


Figure 3.2: Communication layers for the Small JavaSpace Client implementation

Figure 3.2 shows that on the client side, a new small layer, the Small JavaSpace Client, replaces the three layers JavaSpace, Jini and RMI from figure 3.1. The Small JavaSpace Client offers JavaSpace commands to the application in the same way as the standard Java library does by providing a class AbstractEntry with operations for reading, writing and taking entries. However, it does not communicate directly with a JavaSpace via RMI, but communicates with the proxy via TCP/IP (see figure 3.2).

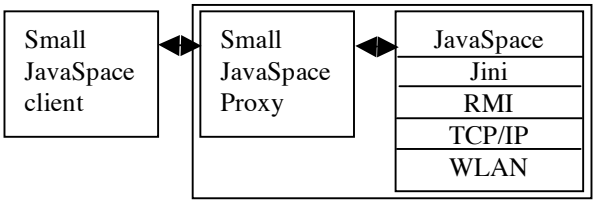


Figure 3.3: Communication between Small JavaSpace Client, proxy and the JavaSpace

Figure 3.3 shows that the Small JavaSpace Client communicates with the JavaSpace via the proxy. The proxy and the space may be run on the same device, so that no communication via Wireless LAN is necessary between them.

As discussed above, entries are objects and thus cannot be sent directly via sockets because of missing serialization, but have to be converted to streams. Entry-to-stream conversion is done by extracting all information from an entry that is necessary to reconstruct it and sending this information as a stream to the receiver. That requires that all entry information (name, variables, variable values) must be displayable as a stream of bytes. Furthermore this conversion is based on the assumption that the receiver already knows the entry’s class. Thus the receiver can create a new instance of this entry and give values, which are taken from the stream, to its variables. These variables are objects themselves, so their values must be representable as arrays of bytes, or easier: strings.

All in all the client / proxy architecture allows writing, reading and taking entries from a small device running MIDP with the same interface as when running a full JVM is run. Still some limitations of this architecture exist:

Notification only works in one direction. When writing an entry to the JavaSpace, a device running a full JVM can be notified about this entry. The other way around is not supported because this would have required RMI support on the small device. Furthermore only basic lease operations are available and transactions are not implemented at all. But full integration of these properties is not excluded by the architecture. Lastly, fields are currently restricted to simple types, and serialization code has to be provided manually. However, this

can be automated and more complex types be supported by the addition of a pre-compiler to the system.

3.2. Implementation

For using the usual JavaSpace abstractions, nearly all classes and interfaces known from standard JavaSpace can written with the same API. All methods of these classes have to be written in a way that entries are transmitted as socket streams between client and proxy. For this the types, names and values of the entry and all its fields are necessary.

In a full JVM this information can be extracted by using the "Class.getFields()" method and then converted to an array of bytes by using the "String.getBytes()" method. As so called "reflected objects" like "java.lang.reflect.Field" are not implemented in MIDP, the information has to be provided by the entry object itself in a special function. This function is called every time an instance of the entry is sent.

Hence, when creating an entry class, the class has to be equipped with a send- and a receive function (for writing and reading / taking entries). These functions have to be created manually by a simple algorithm. This algorithm might also be executed by a pre-compiler and so be hidden from the user. Thus the standard API for JavaSpace commands can be kept. The disadvantage for this is that some pre-processing is necessary.

I.e. the JavaSpace.write() method simply calls the send method of the entry. This send method transmits a stream to the proxy, which contains the following information, separated by commas:

- The operation command done on the entry (e.g. "write" or "takeIfExists")
- The transaction (if not implemented "null")
- The timeout (e.g. "10000")
- The entry's class name (e.g. "myEntry")
- For each variable:
    - Its type (e.g. "String")
    - Its name (e.g. "myString")
    - Its value (e.g. "abc")

If any variable has a value of null, no information is send, neither type nor name nor value. As all extracted information is stored as a string, you can only use variables for entries which can be displayed as strings.

The complete information is stored and edited in string form and only converted to an array of bytes for streaming it to the proxy side. The proxy reconverts the array to a string and extracts the information from it by looking for commas. Then a ClassLoader loads the class given by the submitted class name. For this the ClassLoader must be able to find the according entry class. A new instance of this class is created and its fields are set to the values given by the string or null. By the help of the submitted operation command the operation is finally done on the object.

If an entry is written to the JavaSpace, the lease's hash code is returned to the client to provide a reference to the lease for a later lease renewal.

If the operation done on the entry is "read", "readIfExists", "take" or "takeIfExists", a template is sent and an entry matching this template is returned to the PDA. Similar to sending an entry from the client to the proxy, the entry's information is stored in a string (by help of the Class.getFields() method) and streamed to the client. As the entry's class name is already known and no operation commands are needed anymore, only the variables' names and values are attached to the string. After conversion to an array of bytes, streaming and re-conversion to a string, the variable names are used for the receive function on the client side:

Also on this side a new object is created, the variables are set to the values taken from the string. The object is used as return value for the operation function (e.g. "read"). Thus, by the help of send, receive and the proxy, all entry operation commands can be implemented on an MIDP device.

Furthermore lease operations are handled by the client / proxy implementation. In the same way like entry operations, lease operations are converted via strings to an array of bytes and sent to the proxy. There the lease operation information is extracted and from this the operation is executed. The correct lease is found via its hash code which was given to the client after writing an entry to the JavaSpace (see above).

4. Results

4.1. Measurements

The implementation of the Small JavaSpace Client shows three main differences to the standard Jini implementation: it needs fewer communication layers and less data to be sent via wireless communication, as it transmits Strings instead of complete objects. Moreover, the Small JavaSpace Client can be used with small devices with MIDP instead of standard PCs with a full Java Virtual Machine. All those facts raise the question how the performance changes compared to the usual Jini implementation.

In order to measure this performance change, several runtime test were made. These tests bring out the timing differences while writing, reading and taking entries with both the Small JavaSpace Client and standard Jini. The Small JavaSpace Client is run on several devices: A Palm Tungsten C, a HP iPAQ h5500 and a usual PC, an IBM ThinkPad X31.

	Palm Tungsten C	HP iPAQ h5500	IBM ThinkPad

	Palm Tungsten C	HP iPAQ h5500	IBM ThinkPad
Operating System	Palm OS 5.2.1	Pocket Windows 2003	Windows XP Professional
Virtual Machine	IBM J9 Virtual Machine	IBM J9 Virtual Machine	Java Virtual Machine 1.5.0
WLAN	802.11b	802.11b	802.11b
Processor / MHz	Intel XScale 400MHz	Intel XScale 400MHz	Intel Pentium M 1400 MHz
Memory	64 MB	128 MB	512 MB

Table 4.1: Overview of devices used for the tests

The JavaSpace is started on an IBM ThinkPad. As the Small JavaSpace Proxy is an extension to this JavaSpace, it is executed on the same host. The runtime tests are generally started on other devices. On both PDAs this is a usual MIDP application extending the Small JavaSpace Client implementation, on a second IBM ThinkPad two different tests are carried out:

1. The Small JavaSpace Client files are integrated into a Java project which does not contain the Jini libraries. The socket and the Input- and OutputStream commands are changed from the MIDP API to the Java API. Based on this the same runtime test is started as on the PDAs. The test results are shown in the "SJS Client" column of tables 4.2 and 4.3.
2. Another program simply uses the standard Jini / JavaSpaces libraries, finds the JavaSpace via a lookup and reads, takes or writes entries. Its results are shown in the "Full Jini" column of the two tables.

Each test is run 13 times. In each of these 13 runs 100 times the same task is executed. The first three runs are neglected because of the cache warming effect [19]. From the other 10 tests the average is taken and presented in tables 4.2 and 4.3.

Test No.	Palm	iPAQ	SJS Client	Full Jini
1	3893	1775.6	812.2	1292.9
2	4430	2081.3	957.3	1582.3

Table 4.2: Runtime tests 1 & 2 results (all values in ms)

Test 1 shows the results for writing and taking 100 entries. Each entry consists of four fields, two strings and two integer objects. The resulting string, that is sent to the proxy (see chapter 3.2), contains 108 bytes. In the full Jini implementation only the serialized entry (without the header information from the different layers) contains already 236 bytes.

In this test to things are noticeable: The Palm device is more than twice slower than the PocketPC, although both devices nearly have the same hardware capabilities. The PocketPC itself is more than twice slower than the PC. Moreover the test shows that the Small JavaSpace Client implementation on the PC is 37% faster than the runtime test of the standard entries.

Test 2 shows the results for again writing and taking 100 entries. This time the entry contains more data. So the string, that is sent, contains 156 bytes. The results show that the time it takes to send those longer messages increases significantly, but the relations between the devices stays the same.

Test No.	SJS Client	Jini
3	516.8	455.7
Difference 1-3	295.4	837.2

Table 4.3: Runtime test results (all values in ms)

For getting information about the communication speed of the Wireless LAN connection, the same runtime test as for test 1 is executed, but this time the runtime test program is executed on the same device as the JavaSpace. The results for this test are shown in table 4.3. In these results the time needed for communication via Wireless LAN is excluded. To get this excluded time, you need to subtract the values from test 3 from the according values from table 4.2 (812.2 – 516.8 and 1292.9 – 455.7). The results are presented in the next row (295.4 and 837.2) and show that the Small JavaSpace Client's time needed for communication is nearly three times faster than the time for the Jini implementation.

Based on the results from tables 4.2 and 4.3 several conclusions can be drawn. The most important result is that the Small JavaSpace Client implementation is faster than the standard Jini implementation while writing, reading or taking the same entries. This can be directly seen in table 4.2, where the Small JavaSpace Client executes the same task on the same device faster than the full Jini / JavaSpaces implementation. But if we compare the figures shown in table 4.3, we see that the local implementation Small JavaSpace Client is slower than the local Jini / JavaSpaces implementation.

This implies that the derived networking overhead of the full Jini version is significantly larger than of the Small JavaSpace Client. Reasons for this might be that the Small JavaSpace Client just skips several layers of the middleware (see Figures 3.1 and 3.2) and that network messages produced by the Small JavaSpace Client are smaller than those of the full Jini which relies on Java object serialization (e.g. 108 bytes versus 236 bytes in our example). Furthermore, relatively small value of 455.7 ms in the local case for the full Jini can be caused by optimizations built-in in RMI for local invocations, thus yielding the high derived value of 837.2 ms for the Jini networking overhead.

A second important conclusion concerns the hardware used for the runtime tests. The test results prove that the execution speed of a device is not only dependent on the processor power.

In table 4.1 it was shown that the two PDAs have the same processor with 400 MHz, while the PC's Centrino has a 3.5 times higher time rate. All three devices communicate via 802.11b. Nevertheless the results show that the PC is twice as fast as the PocketPC, which itself is twice as fast as the Palm.

One possible reason for this might be the different hardware architecture of the devices. At least it is obvious the there is a great difference in the memory of the PocketPC and the Palm. As the 51 MB of free memory of the Palm should be far enough to run the Small JavaSpace Client implementation as well as with 128 MB, this difference can be found in the speed of the memory. While the PocketPC is using fast SD-RAM, the Tungsten uses the same slow flash memory for fixed data as for random access.

So we can conclude that an ideal device for running the Small JavaSpace Client implementation is not only based on a fast processor. Also other hardware properties, like the architecture or fast RAM, are important for a good performance. Moreover the different implementations of the virtual machine and of the operating systems on the different machines most probably have performance impacts, but a further investigation here is outside of the scope of this paper.

4.2. The Rent-A-Robot Application

For educational purposes, a scenario called Rent-a-Robot has been set up [18]. This scenario works on Lego robots with different capabilities. When a user wants to rent a special robot and make use of its capabilities, he needs to get access to this robot.

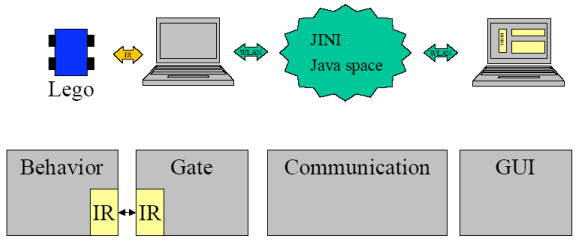


Figure 4.1: Rent-a-Robot communication scheme

For this, a Java / Jini network is built up, which consists of a server and several clients. These clients are users and robots. The robots are connected via infrared to their docking station, which includes the Lego infrared tower and a gate to the JavaSpace (see figure 4.1). Via this gate information can be exchanged with the server. The users on the other side use a GUI, which is also connected to the JavaSpace, as interface to the robots.

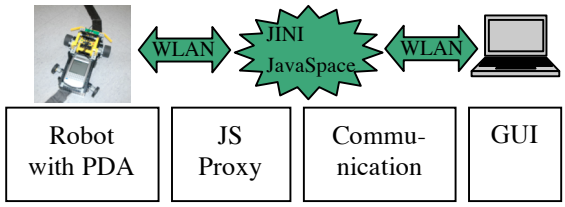


Figure 4.2: Rent-A-Robot with communication scheme

In this setup one major problem occurs. The robot is not always in range of the docking station. So, a PDA is mounted on the robot, replacing the gate with its infrared port and its Wireless LAN module. The PDA is not able to run a complete Java Virtual Machine, so a K Virtual Machine is installed on it. On this K Virtual Machine a MIDlet containing the Small JavaSpace Client is running (see figure 4.2). The Small JavaSpace Proxy resides on the same device on which the JavaSpace is started. The client is connected to the proxy via the proxy's IP address.

First of all the robot needs to be registered to the server. For this the PDA writes an entry containing the robots ID and its capabilities to the JavaSpace via the proxy. The entry is read by the server and the robot is put to the list of available robots. The user on the other side uses a GUI to get this information. Thus he gets access to the robot.

If the user wants to rent this robot and give a task to it, he writes an entry to the JavaSpace via the GUI. Via a corresponding template this entry is read by the PDA (again via the proxy) and the entry's information is forwarded to the robot. When the task is finished, the robot sends a message to the PDA. This information is forwarded to the server and thus to the user via a JavaSpace entry. After that a template is written to the space waiting for a new task.

5. Conclusions

This paper presents a middleware for the integration of small devices into a JavaSpaces environment. This middleware consists of a distributed architecture, a client (Small JavaSpace Client) on a small device and a proxy running on a full Java Virtual Machine, communicating via socket connections. The client is equipped with classes and interfaces identical to those from the real JavaSpace, but all JavaSpace commands (e.g. write, read, take, lease operations …) that are executed on the Small JavaSpace Client result in a message to the proxy. This message contains all necessary information to enable the proxy to execute this same command on the full Java Virtual Machine. The performance measurements show that the Small JavaSpace Client performs faster than a standard Java virtual machine.

Advantages of this architecture are the fast performance and that MID profile and the JavaSpace API are

kept. Thus, no additional programming effort is required on the application level. Disadvantages are that the proxy location must be known to the client and that some additional pre-processing of the send and the receive method is required, either manually or by a pre-compiler. Moreover entries can only contain variables of simple types.

References

[1] Sun Microsystems, *The Jini Technology Surrogate Architecture Overview*, Technical report, Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303 USA, 2001.

[2] J. Gosling, H. McGilton, *The Java™ Language Environment A White Paper*, JavaSoft 2550 Garcia Avenue Mountain View, CA 94043 U.S.A, 1996.

[3] H. Wong, *Developing Jini Applications Using J2ME Technology*, Pearson Education, 1st Edition, 2002.

[4] Inca X. *Inca X ME for MIDP and Jini Technology*, www.incax.com.

[5] The Open Services Gateway Initiative. *OSGi Service Platform Release 3*, IOS Press, Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands, 2003.

[6] The Open Services Gateway Initiative. *About the OSGi Service Platform*, OSGi Alliance, 2004.

[7] H.-P. Bischof, A. Kaminsky, *Many-To-Many Invocation: A New Object-Oriented Paradigm for Ad Hoc Collaborative Systems*, Technical report, RIT IT Lab, 2002.

[8] H.-P. Bischof, A. Kaminsky, *New Architectures, Protocols, and Middleware for Ad Hoc Collaborative Computing*, Middleware 2003 Workshop on Middleware for Pervasive and Ad Hoc Computing, Rio de Janeiro, Brazil, 2003.

[9] PsiNaptic Inc, *JMatos Jini Network Technology for Embedded Processors*, 2001.

[10] A. Kaminsky, *JiniME: Jini Connection Technology for Mobile Devices*, 2000.

[11] YoSpace, *YoCOA: A Lightweight, Bandwidth Efficient Distributed Objects Package for Mobile Applications*, www.yospace.com

[12] C. Bock, *Zugriff auf Jini-Dienste mit MIDP*, Diplomarbeit, Hochschule Harz, 2002.

[13] M. Wejrot, F. Hederstierna, *Jini in a Cellular Telephone*, Master's Thesis, Lund Institute of Technology, 2000.

[14] A. Kaminsky, *Running Jini Network Technology in Small Places*, Information Technology Laboratory Rochester Institute of Technology, 2000.

[15] Sun Microsystems, *Jini Architecture Specification, Version 2.0*, Sun Microsystems, 4150 Network Circle, Santa Clara, CA 95054 USA, 2003

[16] Sun Microsystems, *Java™ Remote Method Invocation Specification*, Sun Microsystems, 901 San Antonio Road, Palo Alto, California 94303, USA, 2002

[17] Sun Microsystems, *JavaSpaces™ Service Specification*, Sun Microsystems, 4150 Network Circle, Santa Clara, CA 95054 USA, 2003

[18] M. Mock, W. Nowak, Teaching Middleware with mobile Robots, submitted to the Workshop on Robots in Computer Science Education, GI-Tagung, Bonn 2005.

[19] Intel Corporation, *Using the RDTSC Instruction for Performance Monitoring*, Intel Corporation, 1997

[20] Sun Microsystems, *Mobile Information Device Profile (JSR-37)*, Sun Microsystems, 901 San Antonio Road, Palo Alto, California 94303, USA, 2000

[21] N. Carriero and D. Gelernter, "Linda in Context," Communications of the ACM, vol. 32, no. 4, pp. 444-458, 1989.