Semester Thesis

Ad-Hoc Services

Marc Schiely, Clemens Schroedter
maschiel@student.ethz.ch, clschroe@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich
Winter 2002 / 2003

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Advisor: Aaron Zollinger

Contents

Chapter 1

Introduction

Marc Schiely

The objective of this thesis was to implement some basic ad-hoc services which can be used by Java applications running on the system. To realize this task, we needed a framework which allows applications to access the functions of the services. The services should be started at the start-up of the machine and run in the background.

One of the demands was that the applications should not need to know anything about the kind of communication that is used between the application and the service. It should be possible to add new services to the existing framework, too. Another requirement was that it is possible to easily exchange the communication technology without changing services and applications. Chapter 3 describes the architecture of the framework we designed for those demands. Chapter 4 describes the compiler which is used to add new services to the machine.

Another aspect we paid attention to was performance. For selecting the right communication, we measured different kinds of communication for the messaging between the two Java Virtual Machines (Application - Services). In chapter 2 the results of these measurements are described.

Chapter 2

Performance Measures

Clemens Schroedter

2.1 Testing what?

To find the best communication between the different Java Virtual Machines we first considered the internet to get an overview of the possible communication technologies. Basically we found the possibilities of using RMI (Remote Method Invocation) or Corba. We also implemented our own communication layer using sockets.

2.2 Testing communication with a small amount of data

The next question was how to test the different communication methods. Therefore, we implemented a very simple Calculator which we used to test the performance for a small amount of data. First we established the communication and then we called the remote procedure 10 times. The Calculator received 2 integers and passed back one integer via the tested communication layer.

2.3 Testing communication with a big amount of data

The next test we performed was about the speed of transmission of large data. Therefore we sent in each call an array of 10'000 bytes.

2.4 Conclusion

Looking at the data, it was obvious to us not to use Corba. We were able to set up the fastest communication with our own implementation. But designing and implementing an own communication layer would have exceeded the scope of this thesis. One more reason against implementing the communication layer by ourselves was, that the own final implementation would probably be slower than the communication layer we tested. Thinking that the main effort should lie in the ad-hoc services and the framework design, we decided to use the RMI solution.

RMI													
measurement	Lookup	Call 1	Call 2	Call 3	Call 4	Call 5	Call 6	Call 7	Call 8	Call 9	Call 10		average Call
1	811	130	120	90	81	110	100	100	110	100	101		104,2
2	791	100	110	91	110	100	90	110	110	90	111		102,2
3	751	90	110	111	100	90	90	100	110	90	100		99,1
4	731	90	100	91	80	100	80	110	90	90	91		92,2
5	731	80	120	80	80	90	90	91	100	100	80		91,1
average	763	98	112	92,6	90,2	98	90	102,2	104	94	96,6		97,76
selfmade													
measurement	Init of sockets	Call 1	Call 2	Call 3	Call 4	Call 5	Call 6	Call 7	Call 8	Call 9	Call 10		average Call
1	150	190	40	20	31	30	20	40	20	30	20		46,78
2	151	181	40	20	20	30	30	30	20	20	20		43,44
3	151	181	40	20	30	20	30	30	20	20	30		43,44
4	140	180	30	30	31	30	20	40	20	30	20		45,67
5	151	181	40	20	20	30	20	30	20	30	20		43,44
average	150,5	185,5	40	20	25,5	30	25	35	20	25	20		45,11
CORBA													
measurement	Init	Call 1	Call 2	Call 3	Call 4	Call 5	Call 6	Call 7	Call 8	Call 9	Call 10		average Call
1	500	1302	401	400	391	401	410	401	400	381	370		485,70
2	451	901	391	380	461	421	400	391	370	401	380		449,60
3	451	881	360	361	370	361	390	361	371	360	371		418,60
4	451	861	360	351	360	361	370	351	370	361	370		411,50
5	451	871	351	350	361	360	381	350	361	360	351		409,60
average	460,8	963,2	372,6	368,4	388,6	380,8	390,2	370,8	374,4	372,6	368,4		435,00

Figure 2.1: Measurements with small amount of data, implementation in Java, times in ms

RMI												
measurement	Lookup	Call 1	Call 2	Call 3	Call 4	Call 5	Call 6	Call 7	Call 8	Call 9	Call 10	average Call
1	731	40	20	50	30	30	40	30	20	20	30	31
2	761	30	20	20	30	30	50	30	20	30	20	28
3	721	40	30	30	30	30	40	20	20	30	50	32
4	731	30	20	20	20	30	60	30	60	20	20	31
5	721	30	20	20	20	30	40	20	30	30	60	30
6	721	20	30	20	20	20	40	20	20	20	10	22
7	711	30	20	20	20	30	40	20	30	20	20	25
8	731	40	31	20	40	30	40	20	20	20	20	28,1
9	731	20	20	20	20	20	40	20	20	20	21	22,1
10	731	20	20	20	20	21	40	30	20	20	20	23,1
average	**729**	**30**	**23,1**	**24**	**25**	**27,1**	**43**	**24**	**26**	**23**	**27,1**	**27,23**

selfmade												
measurement	Init of sockets	Call 1	Call 2	Call 3	Call 4	Call 5	Call 6	Call 7	Call 8	Call 9	Call 10	average Call
1	150	0	10	10	0	10	10	0	10	10	0	6,67
2	151	0	10	0	0	10	0	0	10	0	10	3,33
3	150	0	0	10	0	0	10	10	0	10	10	4,44
4	150	0	10	10	0	10	10	0	0	10	10	5,56
5	150	10	0	10	10	10	0	0	10	0	10	5,56
6	151	10	0	0	10	0	0	10	0	10	10	4,44
7	151	0	10	10	0	10	10	0	20	0	0	6,67
8	141	0	0	10	0	0	10	0	0	10	10	3,33
9	150	0	10	10	0	0	10	0	0	10	10	4,44
10	151	10	0	10	0	10	10	0	10	0	10	5,56
average	**149,5**	**3**	**5**	**8**	**2**	**6**	**7**	**2**	**6**	**6**	**8**	**5**

CORBA												
measurement	Init	Call 1	Call 2	Call 3	Call 4	Call 5	Call 6	Call 7	Call 8	Call 9	Call 10	average Call
1	581	641	60	70	60	111	60	70	80	80	70	130,20
2	460	411	80	70	60	70	61	70	80	80	70	105,20
3	450	421	70	50	50	90	60	61	70	70	70	101,20
4	450	401	50	50	50	80	70	60	101	60	60	98,20
5	451	411	60	50	50	70	60	50	70	80	121	102,20
6	451	410	61	50	60	70	60	70	70	50	60	96,10
7	451	411	50	50	60	70	50	50	60	60	61	92,20
8	450	401	60	60	70	171	50	40	60	50	60	102,20
9	450	401	60	50	50	70	40	50	71	60	60	91,20
10	450	401	50	60	50	70	50	50	81	60	70	94,20
average	**464,4**	**430,9**	**60,1**	**56**	**56**	**87,2**	**56,1**	**57,1**	**74,3**	**65**	**70,2**	**101,29**

Figure 2.2: Measurements with big amount of data, implementation in Java, times in ms

Chapter 3

A Framework for System Wide Accessible Services

Marc Schiely

3.1 Demands

For the design of the framework we had to fulfil various demands. Beside performance and platform independence, it was important that the communication technology used in the framework could easily be exchanged. The application should not know about the way it communicates with the service. This problem was solved by the introduction of stubs. The task of the stubs is to abstract from the used kind of communication technology.

Another demand was that new services could be plugged into the system without rewriting neither the framework nor the applications. The new service also should not know about the kind of communication technology. This abstraction was done by writing a compiler which generates the needed files for the new service with RMI-specific code inside.

The implemented services should run in the background. At the start-up of the machine, the services that are frequently used should be started. This way they would be accessible to the applications without additional delay when first used.

3.2 Architecture

Because of the measures we made, we decided to implement the framework using RMI. Figure 3.1 shows the architecture of the framework.

First the Servicer must be started (1). It binds the Registry to a name so that it can be accessed by RMI. An application which wants to use a service needs the object AH-Stub. This stub provides a method getService(String) which can be used to get the service which is correlated to the service name (2).

If the service is not yet started, the registry starts it and binds it to the name which is found in the file Services.txt. For further requests for this service, the registry returns the name of the implementation of the service.

With the name the AH-Stub gets from the registry, it can create an RMI-Stub of the service. This is returned to the application. Now the application can call methods

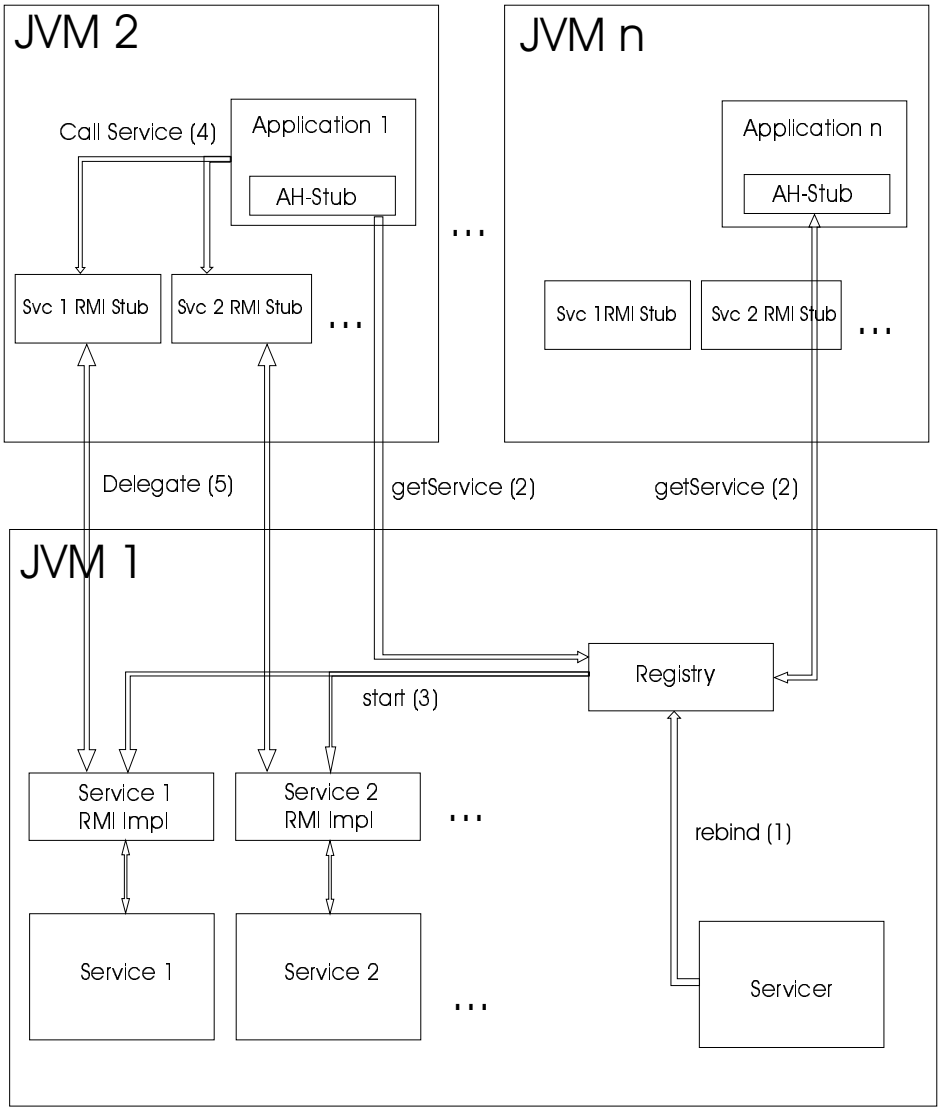


Figure 3.1: Framework for System Wide Accessible Services

as they are defined in the interface (4). The calls are delegated from the RMI-Stub to the RMI-Implementation (5) which calls the method on the real implementation and returns the result to the application.

3.3 Changing the Communication Technology

The following classes have to be newly implemented, in order to exchange the communication technology used.

- Servicer: still starts the registry and makes it accessible to the AHStub.
- AHStub: provides a method getService(String) which returns an instance of a service.
- Registry: the interface between the services and the AHStub.
- AHSCompiler: if needed, a new compiler has to be written. This has to generate the files containing the specific code for the communication technology.

The applications using the services have to use the new AHStub. The service files specific to a communication technology must be adapted.

3.4 Open Problems

The framework does not allow any callbacks. Therefore the services can not call methods on applications. This may be implemented in a future work.

Chapter 4

Service Compiler for RMI Framework

Marc Schiely

4.1 Purpose of Compiler

The demand of not knowing about the communication technology leads to the requirement that the service should not contain any RMI-specific code. So we need a compiler that inserts the code needed into the classes and generates the stubs that are used for the communication between applications and services.

The so-called AHSCompiler takes the interface and the name of a new service as its input and generates three needed classes and a skeleton for the implementation of the service.

4.2 Description of Finite State Machines

Figure 4.1 contains the schemas of the finite state machines that are used in the compiler.

- The FSMInterfaceScanner is used to generate the interface with RMI specific code. This interface is needed by the RMI compiler.
- The FSMStubScanner generates the stub, which is used by the applications to communicate with the services.
- The FSMSkeletonScanner generates a skeleton for the implementation of the service which is compiled.

4.3 Usage of Compiler

Because the compiler generates files and has to put them into the right directory, it must be run from the directory where the package ad-hoc is placed.

The parameters to start the compiler are the following:

- Fully qualified interface name of the new service.

- Name of the implementation class file without package path – if an implementation does not exist, then a skeleton is built with the specified class name.

The new service has to be registered in the file Services.txt in the directory "adhoc". The form of the entry must be: ServiceInterfaceName=ImplementationClassName

- ServiceInterfaceName must be the fully qualified Java name of the service interface.
- ImplementationClassName must be the class name of the implementation without the package name.

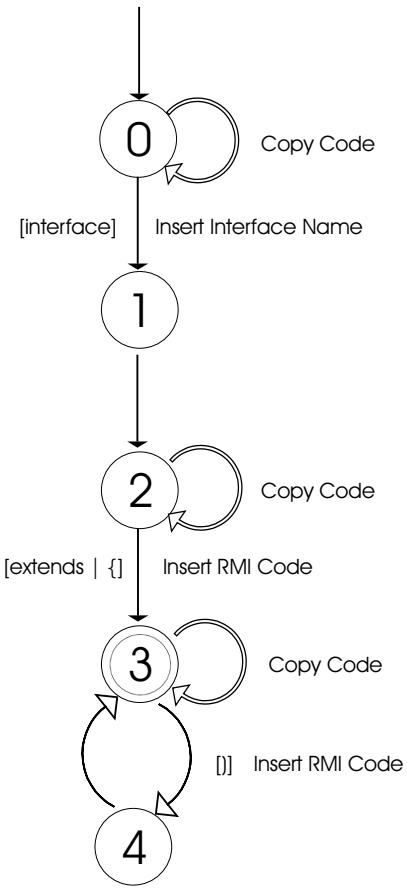
If the service should be started at the start-up of the Servicer, then it must also be registered in the file BasicServices.txt. The form of this entry must be: ServiceInterfaceName=1

- ServiceInterfaceName again has to be the fully qualified Java name of the service interface.

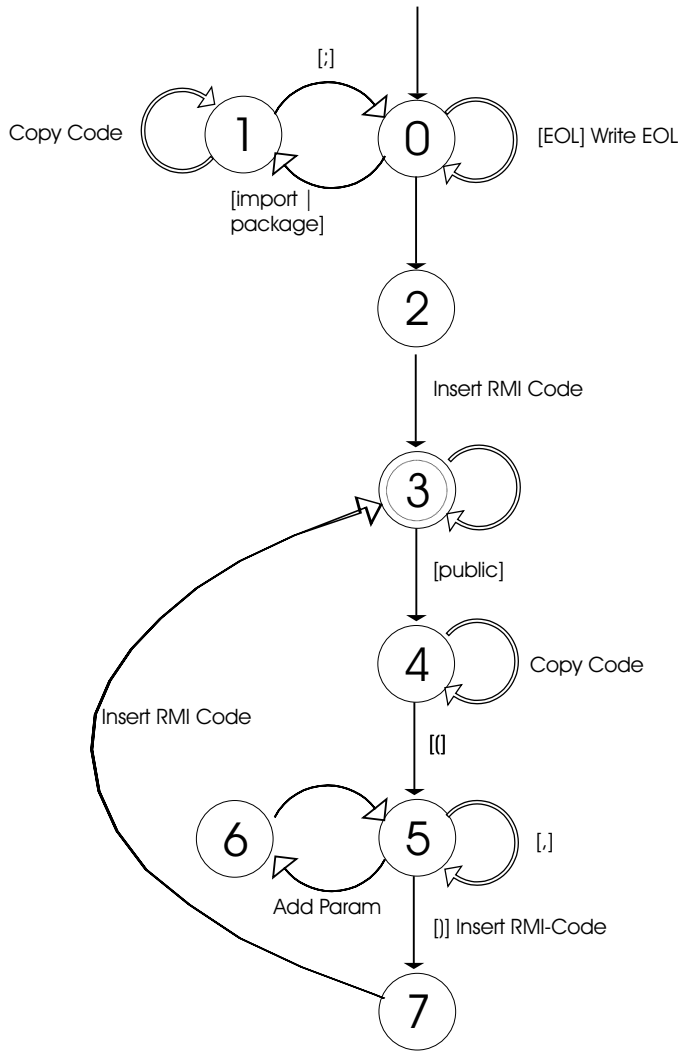
4.4 Requirements of Compiler

- The compiler uses rmic and javac, which must be installed on the machine.
- It is important that each service provides a constructor with exactly one parameter, namely an Integer representing the ad-hoc address of the machine.

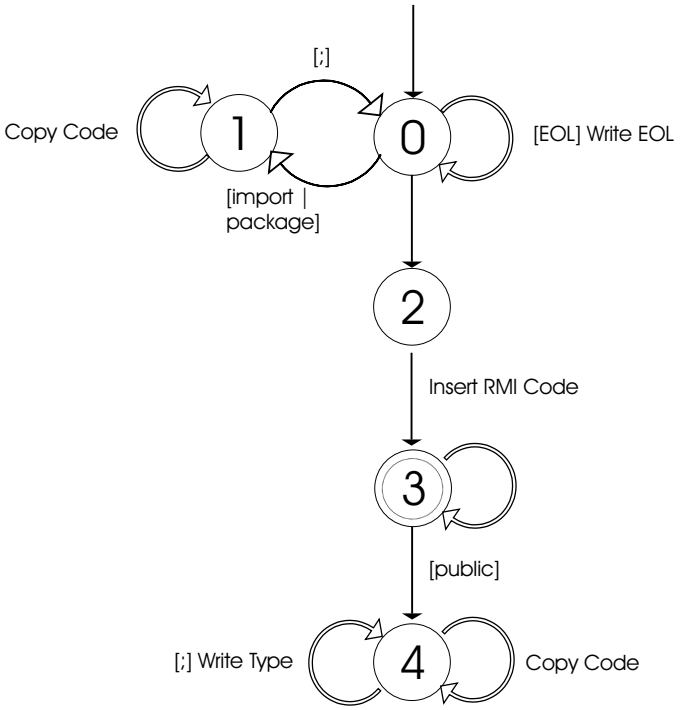
FSMInterfaceScanner



FSMStubScanner



FSMSkeletonScanner



Chapter 5

Implementing new Services

Marc Schiely

5.1 Requirements of Service

The services must follow some guidelines:

- Each service must implement an interface which extends the interface AdHocService.
- A constructor with exactly one parameter, namely an Integer representing the ad-hoc address of the machine must be provided.

5.2 Implementing a new Service

First the class name of the implementation has to be entered into Services.txt. After writing the interface for the service the files must be compiled with the AHSCompiler. If an implementation was provided before the AHSCompiler was run, then no further compilation is needed. Else a skeleton for the implementation is generated which can be used.

More than one implementation can be used. When changing the implementation of a service, the name of the implementation has to be entered in the file Services.txt. Afterwards, the service must be recompiled with the AHSCompiler. No more changes are needed.

To start the Servicer which provides the services, use the file "Servicer.bat" on Windows platforms or "Servicer" on UNIX platforms. These files are located in the directory "adhoc".

5.3 Using Services in Applications

A service can be used in any application by calling getService(String) on an instance of AHStub which encapsulates the registry mechanism of the employed communication technology. The parameter is the fully qualified java name of the interface of the service. On the instance returned, method calls can be made as usual.

Chapter 6

Ad-Hoc-Services

Clemens Schroedter

6.1 General design

The class AHSocketImpl extends the class AdHocNotifier and implements the socket for our ad-hoc services. The AHSocketImpl has an inner class, the ReceiverThread. This Thread is listening to the AHSocketImpl. Each service has to implement the interface AdHocService. If the service should receive anything, the service needs a Listener which implements the interface AHSocketListener. This Listener then has to be registered with AHSocketImpl.addAHSocketLister(AHSocketListener). Once registered, the Listeners are getting notified if a packet with the same type as returned by the method AHSocketListener.getType() has been received. The packets should be generated with the PacketFactory. In this class all packet and listener types are defined. There are two different packetformats: PacketImpl, MHPacketImpl. The PacketImpl is for sending a packet in the single hop modus. The MHPacketImpl is for sending a packet in the multiple hop modus.

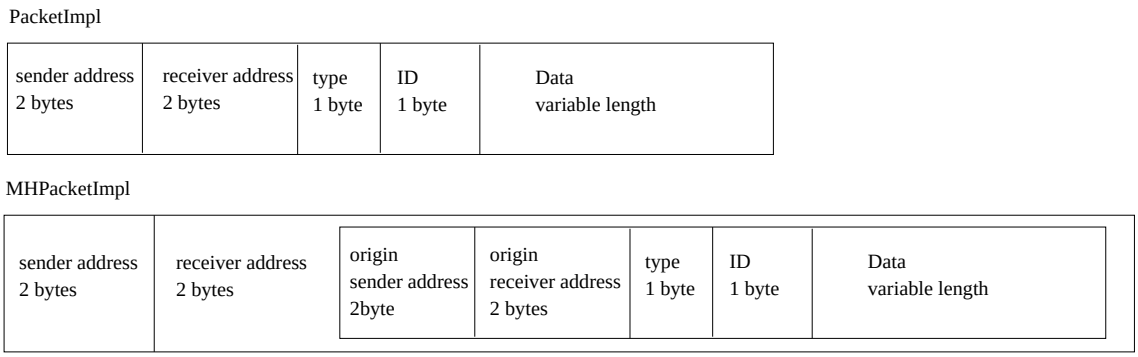


Figure 6.1: The format of single and multiple hop packets, implemented by PacketImpl and MHPacketImpl

6.2 Basic Ad-Hoc-Services

6.2.1 Neighborhood service

The task of a neighborhood service is to deliver the actual neighbors. By actual neighbors we mean the communication partners which can be reached directly, they are all in the range of the WLAN card. Therefore the neighborhood service must run a thread in the background to be up to date. This service is a basic service.

NHServiceImpl1

The neighborhood service uses 2 classes, NHActivator and NHListener to check the environment. The NHActivator starts the process of discovering the area with sending pings. Pings are sent every 3 seconds by default. This can be changed with the method NHService.setBound(int ub). The NHListener registers neighbors in the Neighbour class and answers a ping with a pong. The Neighbour class removes old neighbors, if there was no ping or pong heard within a specified time. The time can be set with the method NHService.setRuntime(long time); the default time is 10 seconds.

NHServiceImpl2

This is a dummy testing service. It returns the Vector [1,2].

6.2.2 Forwarding service

The nature of this service is absolutely reactive. If a multihop or flooding packet is received, this service has to ensure that the packet is forwarded. To prevent that a packet is being forwarded multiple times, it is necessary that the service remembers all packets it forwarded once. With this information, it can distinguish whether or not it has to forward this message. The identification of a packet is made by the tuple (OriginSenderAddress, packetid). The tuple window has a size of 1'000 entries.

ForwardingServiceImpl1

The ForwardingListener forwards the packet, if it is for somebody else and the hop counter (TTL) is not zero. If a packet has been forwarded, the hop counter is decreased. The packet is forwarded as a MHPacketImpl. These packets are sent via broadcast, this means the receiver address is set to zero. The TTL can be set in the class Hops. This information will statically be used.

ForwardingServiceImpl2

The ForwardingListener2 forwards the packet if it is for somebody else and if the hop counter (TTL) is not zero. If the OriginReceiver is directly reachable, the packet is being unpacked to a single hop packet (PacketImpl) and is directly sent to the OriginReceiver. If the OriginReceiver is not in the neighborhood, the packet is being forwarded as MHPacketImpl. To know whether the OriginReceiver is directly reachable or not the neighborhood service is being used.

6.2.3 Sniffer service

It may be of interest to know who has been reachable during the time we run the AdHocServices. Therefore, this reactive service sniffes all received packets. By knowing all sender and OriginSender addresses, we know everybody who was ever reachable. Of course this service has to be started when the AdHocServices are started to be sure that all addresses are sniffed.

SnifferServiceImpl1

All sniffed addresses are returned as a Vector by the method getAllSniffedAddresses().

6.3 Non-Basic Ad-Hoc-Services

6.3.1 Flooding service

With this service, it is convenient to send a flooding message. The flooding service may be used by other services.

FloodingServiceImpl1

This service uses a multihop packet, in which there is a flooding packet encapsulated. Receiving a flooding packet, the packet is being unpacked and all waiting services with the same type as the unpacked message will be notified.

6.3.2 Messaging service

The messaging service is thought to be used to exchange messages.

MessagingServiceSHImpl

With this service, the messages are sent in a single hop packet. This is only useful if one knows surely that the communication partner is close enough. To return the message to the application of the receiver, it uses the notifier service.

MessagingServiceSMHopImpl

In this service, multihop packets are being used if necessary. To check if a receiver is reachable within one hop, the neighborhood service is used.

6.3.3 Notifier service

An application can register itself if it implements the interface AdHocApplication. To register, the application has to pass a reference to itself to the service. The application can also unregister passing its id. To get unique ids, the id should be generated with the class ApplicationId. A service, e.g. the messaging service, can call the notify method to pass the received message to the waiting applications. The advantage of this mechanism is, that the application does not have to perform polling. The disadvantage is that the application is now running in the Java Virtual Machine of the Notifier service. A callback mechanism which allows the service to notify the application is not implemented.

NotifierServiceImpl

To keep track of who wants to be notified this implementation uses a Vector to store the corresponding applications.

Chapter 7

Conclusions

We first considered different technologies for communicating between different Java Virtual Machines. We implemented an environment for testing these. Remote Method Invocation (RMI) had the best performance.

The next step was to design a framework for system wide accessible services. A main requirement was to abstract from the communication technology we have chosen (RMI). Therefore, we used the concept of stubs. With this framework it is now possible to change the communication technology without rewriting neither services nor applications. An open problem is the usage of callbacks. In our framework it is not possible that services call methods on applications. This could be done using the same concept as we used for method calls on services. New stubs could be introduced.

We implemented a compiler which makes it possible, to write new services without using RMI-specific code. This makes it easy to integrate new services into the system.

The second half of the semester thesis we identified some important ad-hoc services and implemented them. The most important basic services we implemented are a neighborhood discovering service, a forwarding service and a sniffer service. Afterwards we used these basic services for more advanced services like a flooding service, a messaging service and a notifier service. These can be used in multihop environments.

We focussed on the design and implementation of the framework. Therefore this took us some weeks more than planned. This was also the part of the semesterthesis we learned the most, thanks Aaron.