Pini – A Jini-Like Plug&Play Technology for the KVM/CLDC

Dipl.-Inform. Steffen Deter and Dipl.-Inform. Karsten Sohr

University of Marburg
Department of Mathematics and Computer Science
deter,sohr@mathematik.uni-marburg.de

Abstract. The Jini^{TM} technology provides an infrastructure for spontaneous networking of devices based on Java-classes and - interfaces. With the help of Jini, small devices with minimal resources should be able to join such infrastructures. Due to the fact that Jini is based upon RMI (Remote Method Invocation) and is therefore on top of RMI, it is impossible to Jini-enable these limited devices: the use of RMI by Jini wastes resources which are not available in the aforementioned limited devices. A potential approach to Jini-enable limited devices is to replace the RMI-technology with the RPC-technology (Remote Procedure Call). By means of this technology it is possible to provide an efficient mode of communication for Jini components, i.e. services and their proxies. However it is important to bear in mind, that this technology avoids the major part of resource waste.
A revision of the implementation strategy for this reason and to adapt the Jini technology to the KVM/CLDC is necessary. In the following sections, the integration of an RPC-implementation into the Jini concept and some revised structures of the Jini-implementation strategy will be described.

1 Motivation

The Jini technology [1, 2] which is based upon Java-classes and interfaces [3], provides an infrastructure for spontaneous networking of devices. These devices can be either a service provider or a service consumer. As mentioned in the abstract, often only minimal resources are available for these devices. Yet in this particular field Jini shows great problems: The necessary Jini core components provided by the package "net.jini.core" have only limited functionality. The dynamic spontaneous networking may only be provided by the combination of this core package with the Jini-extension packages. However these extension packages are major resource consumers. Furthermore, the use of the RMI-technology [4] increases the need of resources. Therefore the major goal is a small and efficient implementation of Jini. It is of great importance to avoid using the RMI-technology, yet it is necessary to provide dynamic and efficient spontaneous networking for limited devices.

To achieve this goal, we intend to provide a Jini-like technology – called Pini – that runs on the KVM/CLDC [10].

The testing ground of this implementation will be the PABADIS project (Plant Automation Based on Distributed Systems)[5]: The field of plant automation provides interesting case studies to demonstrate the effect of joining network infrastructures by means of spontaneous networking and agent technology. On this testing ground often only limited devices and/or platforms are available. The term "devices" refers to hardware, which often provides only limited resources in the sense of memory, storage, computational performance, etc. Platform means the available Java platform, e.g., the available JDK version is often less than the JDK 1.2, which is required by Jini [2].

The following sections provide a description of the Pini approach that enables limited devices to use a Jini-like technology for joining networks in a spontaneous manner.

2 Jini Key Components Adapted to Pini

This section provides an overview of some Jini components which have been reimplemented and/or adapted to the Pini-context. Specifically, these components are the discovery and join protocol, the Pini lookup service and its (Pini) lookup service proxy.

2.1 The Pini-Discovery and the Pini-Join Protocol

To participate in a Jini community Jini services and clients have to discover **at least** one lookup service in order to register themselves or to get information about available services in that community (in case of both clients and services). This discovery feature is provided by the discovery protocol. As result of a successful discovery process the service or client performing discovery receives a lookup service proxy. In Sun's lookup implementation this proxy object is submitted as a RMI-marshalled object, and the proxy class will be loaded by the RMI class loader[4]. In the Pini implementation the discovering service/client does not receive a RMI-marshalled object (because RMI is not available), but receives a *ServiceDescription* (see Section "3.3 The Class ServiceDescription") via TCP/UDP, which contains all the necessary information to retrieve the proxy. This essential information contains the URL of the proxy classes and the initial data for this proxy. Now the proxy classes will be loaded by the "Pini class loader" (for details see Section "3.5 The Pini Class Loader") and initialized with the submitted initial data.

To perform discovery on limited devices some modifications of the Jini discovery protocol (as described in the Jini Specification) are necessary: Since it is intended to run Pini on the KVM it is impossible to use multicast for discovery (there is no multicast feature available on the KVM). However, to provide a way for discovering lookup services without any knowledge or preconfiguration the discovery protocol uses broadcast instead. The following figure describes the discovery process again: The client/service starts the discovery process, so that discovery requests are sent out (either via broadcast/multicast or unicast). The lookup service receives these requests and responds with the required information, particularly the URL of the proxy classes and initial data. This information is encapsulated within an object of type *ServiceDescription* (see Section "3.3 The Class ServiceDescription") which can be seen as a kind of "marshaled" (proxy) object. After receiving the response, a discovery event is generated and delivered to the waiting *DiscoveryListeners.*

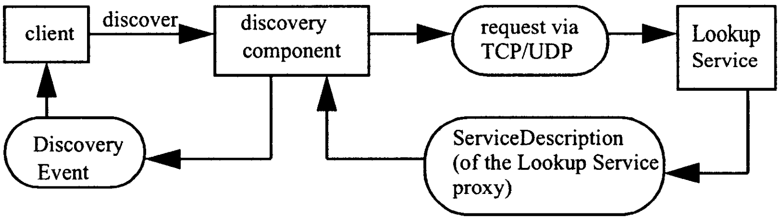

Fig. 1. The (adapted) discovery process in Pini

With the help of the aforementioned lookup service proxies, the hosts (clients/ services) can communicate to the (discovered) lookup services, for instance, to register services or to perform lookup for available services registered at these lookup services. In Sun's lookup service implementation the communication between lookups and their proxies is strongly based upon RMI and the lookup service itself is registered as a RMI remote object at the RMI-Registry.

Such a registration at any *registry* is unnecessary in Pini, because every (Pini-) lookup service proxy knows its "home". The lookup service proxy only has to know the IP-address of its lookup service and the appropriate port number for the communication.

2.1.1 Lookup of Services

The process of looking up services is implemented by calling the `lookup(..)`-methods of the lookup service proxy. Within the proxy implemented by Sun, however, the remote `lookup(..)`-methods of the lookup service are called via RMI, and the services found are returned as RMI-marshaled objects. After extracting the URL of the proxy classes from the marshalled object, the classes of the found services are loaded via the RMI class loader.

In the Pini implementation the lookup service proxy also calls the remote `lookup(..)`-methods of the lookup service, but not via RMI. The lookup service proxy submits the search patterns via TCP/UDP to the lookup-service (as a remote procedure call of the appropriate remote methods). After receiving this remote method call request, the lookup service searches for matching services and resubmits the *ServiceDescriptions* of these found services back to the requesting host. This host generates *ServiceItem* objects, and if the method `lookup(ServiceTemplate, int)` was called, the generated *ServiceItems* are encapsulated in a *ServiceMatches* object. Now, the search result will be returned to the requesting client/service either as the service proxy of the found service (in case of invocation of method `lookup(ServiceTemplate)`) or as a *ServiceMatches* object (in case of invocation of method `lookup(ServiceTemplate, int)`).

Figure 2 depicts this process: A client/service calls a `lookup(..)`-method at the lookup service proxy and provides the search patterns. The proxy submits this information to its lookup service as a remote method call request via TCP/UDP. The search results are returned by the lookup service via TCP/UDP back to the proxy, which generates either the service proxy or a *ServiceMatches* object.

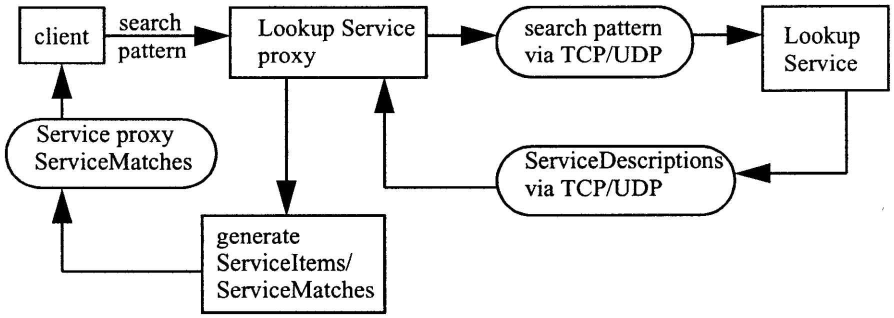
client
search pattern
Lookup Service proxy
search pattern via TCP/UDP
Lookup Service
Service proxy ServiceMatches
ServiceDescriptions via TCP/UDP
generate ServiceItems/ ServiceMatches

Fig. 2. Lookup for services

2.1.2 The Registration of Pini-Services at Pini-Lookups

To be available within a Jini community, services must be registered with lookup services (at least with one) by calling the `register(...)`-method of the lookup service proxy. Within Sun's implementation of the lookup service proxy the remote `register(...)`-method of the lookup service is called via RMI.

In Pini the registration data (*ServiceID*, *ServiceDescription*, attributes, lease duration) are sent via TCP/UDP to the lookup service as a remote procedure call. A further difference between Jini and Pini is: Instead of the *ServiceItem* which is passed to the `register(...)`-method as a parameter, the member fields of that *ServiceItem* are sent to the Pini lookup service. Strictly speaking, a *ServiceDescription* containing all the aforementioned member fields and instead of the service proxy the codebase URL, initial data, etc. will be sent to the lookup. This *ServiceDescription* will be generated by the lookup service proxy via the method `generateDescription()` during the register process. The lookup service receives this registration data and registers the service. As the result of a successful registration process, the lookup service returns the granted lease and the *ServiceID* as a *ServiceRegistration* object back to the lookup service proxy (via TCP/UDP). The proxy in turn returns it to the requesting service.

Figure 3 shows the individual steps of the registration process:

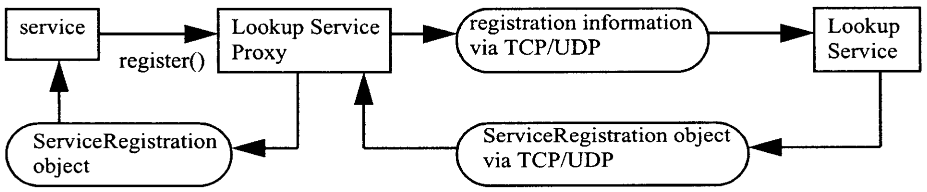
service
register()
Lookup Service Proxy
registration information via TCP/UDP
Lookup Service
ServiceRegistration object
ServiceRegistration object via TCP/UDP

Fig. 3. Registration of a service at a Pini lookup service

2.2 The Pini-Lookup-Service

Lookup services are very important for Jini- and Pini infrastructures, therefore it is necessary to have at least one lookup in every Jini/Pini community. Hence, the resource restrictions of limited devices greatly influence such lookup services also. Yet, the functionality must not be reduced.

A common way to reduce the resource consumption of lookup services is to integrate the RPC-mechanism into the communication between lookups and their proxies. Thus it is possible to avoid the use of RMI for that communication: In Sun's Jini implementation the communication between lookups and their proxies is strongly based upon RMI. Therefore lookup services are registered as RMI-remote objects at the RMI-registry.

By avoiding the use of RMI in the Pini approach, a (Pini-) lookup service need not be registered at a registry like the RMI registry. The entire communication between lookups and their proxies is based upon a Pini-RPC implementation. For details about the protocol see Section "3.4 The Communicator-Package". Moreover, all necessary service-side functionality is described in Section "3.2 Pini-Services" in more detail.

This section only shows the overall integration of RPC into lookup services. Therefore, figure 4 gives the concept of this integration: A lookup service proxy sends a method call request to the lookup. It is then necessary for (lookup service) proxies to know the IP address and the port number of the appropriate lookup. The lookup services will receive the request via the RPC-implementation of the communicator package. This request is decoded by the RPC-mechanism, and the appropriate method within the lookup service will be invoked. The result of such a method invocation is resubmitted to the method caller with the help of the communicator package implementation (exactly: the class *ResultDelivery* is responsible for resubmitting results back to clients).

Examples for communication between lookups and their proxies are: the process of looking up services available in a community (Section "2.1.1 Lookup for Services") or the registration of a service at lookup services (see Section "2.1.2 The Registration of Pini-Service at Pini-Lookups").

2.3 The Lookup-Service Proxy

The main functionality of lookup service proxies is provided by the *ServiceRegistrar* interface of the Jini-specification, however this functionality must not be

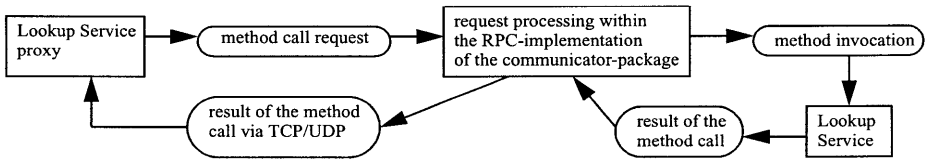
Lookup Service proxy
method call request
request processing within the RPC-implementation of the communicator-package
method invocation
Lookup Service
result of the method call
result of the method call via TCP/UDP

Fig. 4. Communication between a (Pini) lookup service and a lookup service proxy via RPC

reduced. Thus, the major concern here is to optimise the communication between lookup service and proxy. This means that the RPC-mechanism has to be integrated into the communication instead of RMI.

Sun's reference implementation of the lookup service proxy is strongly based upon the RMI-technology. That means that remote methods are invoked via RMI.

In the Pini implementation, however, remote method calls are realized via RPC: the proxy sends out method call requests via TCP/UDP and listens for return values based upon socket servers. These abilities are provided by the communicator package, specifically by the class *Communicator* of this package. To resubmit the result to the requesting lookup service proxy, the lookup service puts it into an object-array. Thus, it is the task of the lookup service proxy to extract the correct return values from that array. An example of the return values extraction from the received object-array is the generation of *ServiceItems/ServiceMatches* within the `lookup(..)` methods of the lookup service proxy (see Section "2.1.1 Lookup for Services"). Therefore the proxy reads the items in the array, casts them into their appropriate types, generates the return values and delivers them to its client. Further details are described in Section "3.1 Service-Proxies and the Class DefaultPiniProxy" and Section "3.4 The Communicator-Package".

Another important modification of the lookup service proxy structure is the extension of the class *DefaultPiniProxy*. This is due to the fact that the proxy needs to know the IP-address of its lookup service and its port for the (remote) communication. By this extension the lookup service proxy inherits the following member fields:

```
public String serviceAddress
public int servicePort
public String codebase
public Class[] interfaces
```

These fields must be initialized by the inherited method `initialize(String, int, String, Class[])` (See Section "3.1 Service-Proxies and the Class DefaultPiniProxy").

3 New Components and Modified Jini Interfaces

This section describes components, that belong to Pini but not to Jini. Moreover, a description of new features that (Pini-) services and proxies require in order to provide the basics for communication via RPC are described. One of these new features is the extension of the class *DefaultPiniProxy* by (Pini-) service-proxies as described in the following section.

The new components include the communicator package "pabadis.kvm.pini.communicator" and the class loader in "pabadis.kvm.pini.util".

3.1 Service-Proxies and the Class DefaultPiniProxy

Service proxies provide their services to clients. The provision of services to clients occurs either with or without communication between the proxies and their services. It is noteworthy to mention, that this communication often occurs via RMI. In Pini the use of RMI technology shall be avoided so that the necessary abilities are provided by the package "pabadis.kvm.pini.communicator". For remote method calls, the method `callMethod(..)` is provided by the class *Communicator*. As parameters this method expects the name of the (remote) method to be called, represented as a string, and an object array containing the appropriate (remote) method parameters. Therefore a method call from a proxy to a service looks as follows:

```
Communicator com = new Communicator(String,int);
Object[] result=com.callMethod(String, Object[]);
```

As the result of the method invocation, the *Communicator* returns an object array containing the necessary information to generate the correct return value(s). The proxy then reads the items from the array, casts them to their appropriate types and generates the expected return value. To obtain the correct value, the communication partners need an appropriate communication scheme. This means that the object in the array should be ordered in a well-known fashion, so both the proxy and the service are able to interpret the value for the method call request. Furthermore, the generation of the appropriate return value from the returned object array must be done by the service proxy; it does not happen automatically!

Another new feature is that **all** service proxies have to extend the class *DefaultPiniProxy*. Hence all service proxies inherit the following fields:

```
public String serviceAddress
public int servicePort
public String codebase
public Class[] interfaces
```

These fields must initialized by the inherited method `initialize(String, int, String, Class[])`. Through the first `String` parameter, the IP-address of the appropriate service is provided to the service proxy. By means of the `int` parameter, the port number will be provided to the service proxy. The second `String` parameter represents the codebase URL of the proxy class file. Finally, the `Class`-array contains the appropriate interfaces, that this proxy implements. To ensure that the array does not contain interfaces, which the proxy does not implement, the lookup service proxy checks this array during the register process. Remember, these array items will be compared to the type values of a given ServiceTemplate during the lookup process.

One may be inclined to inquire, why this is. On the one hand, the extension of the class *DefaultPiniProxy* by Pini service proxies provides a common interface for initializing the downloaded proxy classes, or more concisely, the instance of this particular proxy class. On the other hand, the inherited fields provide the necessary information for using the RPC-protocol, for downloading the class files and for performing lookup. Therefore, this means that the "main" proxy class needs to extend the *DefaultPiniProxy* class, yet not other additional proxy-classes.

3.2 Pini-Services

This section describes the necessary service-side features for the communication between services and their (remote) proxies. If "unlimited" resources are available, the use of RMI is a favourable way to enable remote communication. In the case of limited devices however RMI is not the best solution due to the fact that it is a huge resource consumer.

In Pini an RMI-like registration (at the RMI-registry for example) is unnecessary. To provide remote communication the service must merely ensure that its service proxy knows the IP-address and the port number for communication. This necessary information is put into the *ServiceDescription* of that service. Upon registering the service with lookup services, the *ServiceDescription* is sent to the lookup instead of the whole service proxy. Hence, if clients download the *ServiceDescription*, they obtain all necessary information to load the proxy classes, to initialize the proxy and to communicate back to the service via the proxy. (For *ServiceDescription* see Section "3.3 The Class ServiceDescription", for service registration see Section "2.1.2 The Registration of Pini-Services at Pini-Lookups").

The communication between services and proxies is based upon TCP/UDP. An appropriate mechanism must therefore be provided for such communication. Since the major part of the remote communication between services and proxies is the invocation of remote methods, the following explanation only refers to this interesting field: Specifically, this section describes the necessary service-side conditions for using the (Pini-) RPC-implementation. The integration of RPC into services containing the reception and processing mechanism for method call requests, the method invocation and the resubmitting of the results back to the clients will be highlighted in this section.

As described in Section "2.2 The Pini-Lookup Service", the **reception and processing** of the requests are provided by the classes of the communicator-package: An instance of class *MethodCallListener* listens for incoming requests and generates *MethodCallEvents*. These events are delivered to a *MethodCall-EventHandler* instance. Within this event handler_instance the events are **processed** and the appropriate **methods are called**. The results of these method invocations are **resubmitted** to the clients via class *ResultDelivery*.

Note, that the implementation of the *MethodCallEventHandler* must be done by the programmer. All other necessary features are provided by the package "pabadis.kvm.pini.communicator" (For more details about this package see Chapter "3.4 The Communicator Package")

3.3 The Class ServiceDescription

This class contains not only all necessary information to register the proxy at Pini lookup services, but also information necessary to load and initialize the proxy-object on the client side. In particular, this information includes the service address (IP-address), the service port number, the URL of the proxy classes, the *ServiceID* of the service, the name of the main proxy class, initial data and an array of interfaces that the proxy implements.

The data on the lookup service side are used to find matching services when a `lookup(..)`-method call occurs. Therefore, these data are the initial base for comparison.

On the client side, these data are conversely necessary in order to retrieve the proxy object: The proxy classes are therefore loaded from the URL. The class will then be initialised with the initial data. This action occurs within method `getServiceItem()` which returns a *ServiceItem* object, which contains the proxy-object.

3.4 The Communicator Package

The communicator package provides all necessary classes for the RPC-communication between Pini-services and their proxies. As described in the aforementioned sections, there is a demand for mechanisms to exchange method call requests, and to receive the results. These data exchanges contain the receiving and sending data at both the service proxy side and the service side based upon TCP or UDP. However, there are different mechanisms for services and proxies: The class "Communicator" provides all necessary functionality for sending method call requests and receiving the results (generally speaking at proxy side). Meanwhile, the classes *MethodCallListener*, *MethodCallEvent*, *ResultDelivery*, and the interface *MethodCallEventHandler* provide all necessary functionality for receiving method call requests and sending the results of those method invocations:

3.4.1 The Class Communicator

As previously mentioned, this class provides all necessary functionality for RPC-communication of proxies to their services. This refers to the sending of method

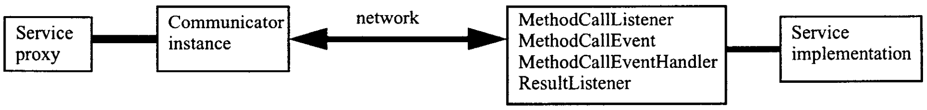

Fig. 5. Overview over the communicator-package

call requests and receiving of results. Only the constructor of the class `Communicator(String, int)` and the method `Object[] callMethod(String, Object[])` exhibit a public interface. For the initialization of this class, the above constructor expects a string representation of the service's IP-address and the appropriate port. On invocation of the method `callMethod(..)` this information is used to establish the communication connection. As parameters the method `callMethod(String, Object[])` expects a string representation of the method identifier and an object array containing the parameters of the method to be called. Within the method `callMethod(..)` a thread will be created that is responsible for receiving the result of the method call request. As the next step, a communication connection to the service will be created (to the listening socket server at service side within a *MethodCallListener* instance) based upon the above information (IP-address and port). With the help of this connection the method identifier, the return port (on which the above thread listens for the result), the length of the parameter array and then the parameters are sent to the service. After receiving the result (through the above thread) it will be returned to the client as an object array. The client is then responsible for generating the appropriate return value from this information. (see Section "3.1 Service Proxies and Class DefaultPiniProxy").

3.4.2 The Class MethodCallListener

The class MethodCallListener is used on the service side to receive remote method call requests. This means that data is received via TCP/UDP-connections. For this reason the class *MethodCallListener* has a thread that listens for incoming method call requests (see previous section), which are buffered in a queue. Thus, the further reception of incoming requests will not be blocked by the processing of such received requests. Processing the buffered requests is done by another thread, named the *ParserThread*. The *ParserThread* determines the source IP-address (for return of results) of the request, reads the return port number (of the listening thread at client side), the method identifier, the number of the parameters and the parameters as objects. With this information a *MethodCallEvent* is generated and passed to the instance of the *MethodCallEventHandler* (which is passed to the *MethodCallListener* as a constructor parameter). After the result is retrieved from the event handler, it is passed to an instance of class *ResultDelivery*, and the results are sent back to requesting client. (Hence every method call request is sent in a "secure" manner because every method call request has a result. If not, the method caller will be informed via a RemoteException!)

Specifically, the class *MethodCallListener* has only two public methods/ constructors: the method `int getPortNumber()` to retrieve the port number (on which the *MethodCallListener* listens for incoming method call requests). The second is the constructor `MethodCallListener(MethodCallEvent-Handler)`. The single argument of the constructor is the event handler that processes the generated *MethodCallEvents*. For an overview of the functionality of the *Method-CallListener* see the figure below:

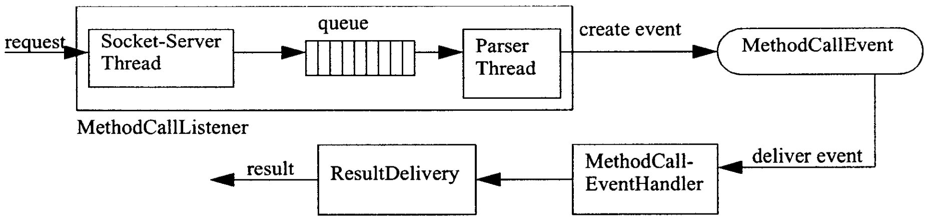

Fig. 6. Overview over the MethodCallListener-class and its functionality

3.4.3 The Class MethodCallEvent

A MethodCallEvent contains all necessary information pertaining to a method call request. In particular, the class contains the following member fields:

```
public class MethodCallEvent{
  private String methodName;
  private Object[] parameters;
  public String getMethodName();
  public Object[] getParameters();
  public MethodCallEvent(String, Object[]);
}
```

It is important to note that this class is not an extension of class java.util.Event-Object, because this package may not be available on limited devices. However, the basics for event processing in the Pini-concept are the same as in the Java event model.

It is not necessary to look at the member fields in detail. As mentioned before, those *MethodCallEvents* are generated within the *ParserThread* of class *Method-CallListener* and passed to the *MethodCallEventHandler*. This event handler is the subject of the next section.

3.4.4 The Interface MethodCallEventHandler

The major task of the *MethodCallEventHandler* is the processing of *MethodCall-Events* that are generated by the *ParserThread* of the *MethodCallEventListener*.

For this event processing the interface contains only a single method: `Object[] parseEvent (MethodCallEvent)`. However, the implementation of this method is up to the service programmer. Hence the information within the event must be interpreted, and the appropriate actions must be implemented: the determination of the method to be called and which parameters this method expects. The parameters are provided by method `Object[] getParameters()` of the class *MethodCallEvent*. After a type cast to the appropriate type the parameters can be passed to the method that has to be invoked. In order to return the result to the client, an object array will be generated and passed to the *ResultDelivery* class. This class is responsible for sending back the result array to the method caller.

3.4.5 The Class ResultDelivery

The primary task of this class is to send the results of a remote method invocation back to the appropriate client. Since the resubmission of results is independent from any other action, the *ResultDelivery* class is an extension of class Thread. Due to this, the processing of further *MethodCallEvents* by the *MethodCallEvent-Handler* is not blocked by the resubmitting of those results. As mentioned in Section “3.4.2 The Class MethodCallListener”, the result of the method call will be passed to the *ResultDelivery* instance and sent to the appropriate client.

Therefore, a communication connection is established via TCP or UDP, which is based upon the IP-address and the port number passed to the constructor `ResultDelivery(String, int)` of this class. The result array is sent to the appropriate client through this connection. After that the thread can be destroyed.

3.5 The Pini Class Loader

Class loaders dynamically load classes if necessary. URL class loaders in particular download classes from a specified URL. The Pini class loader also functions in this manner: In particular, the Pini class loader has to download the class files and write them to a temporary classpath. After that, the appropriate `Class`-object will be created via `Class.forName(name)`. Thus, the *PiniURL-ClassLoader* is akin to a class FILE loader.

Implementing such a class loader is necessary because the KVM does not provide user defined class loaders, however proxy classes must be downloaded from their codebase. When the client shuts down or starts up the computer, all classes in the temporary classpath are deleted.

4 State of the Art and Results

This section descibes the current state of the art of the Pini technology and some results in comparison to Jini.

At the time of writing this paper the following features are available. The lookup service is implemented and works fine. Services and clients can perform

Table 1.

Jini core	32 kB
Jini extension	230 kB
Lookup Service	501 kB
RMI	401 kB
Summary	**1167 kB**

discovery and lookup. Services can register themselves with lookup services, leases will be granted by the lookup service for service and remote event registration. Additionally, the mechanisms for using attributes and adminstration are currently in development.

Table 1 shows the Jini disk space cosumption. In comparision to Jini it is noteworthy that the memory consumption of Pini is much lower: currently Pini needs about 250 kB and there are no other features necessary.

The run time memory consumption of Pini should be determined in the next few months. The comparision of this memory consumption to Jini will be explained in further publications.

5 Related Work

As mentioned in the sections before, the major resource consumer within Jini is the RMI-technology. Therefore, our major goal was to avoid this resource waste and thus to prevent the use of RMI. However, there are several attempts to provide so-called RMI-light technologies, e.g., [8]. Hence, on the one hand it may be possible to replace Sun's RMI technology used within Jini with such a RMI-light technology in order to Jini-enable limited devices. On the other hand, these aforementioned technologies sometimes use Java features that are often not available on limited devices.

An attempt to provide a Jini-like technology for limited devices is described in [6]. But unfortunately, this technology also uses features that are mostly not available on limited devices.

Sun has realized that it is impossible to Jini-enable limited devices. Therefore they recommend their "Surrogate Architecture"[9]. However, this technology cannot be used as a stand-alone Plug&Play technology: It expects a proxy host (the so-called "surrogate host"), which connects the device to the "Jini world".

A very similar technology to Pini is the TINI technology, which was presented on the "Fifth Jini Community Meeting" in December 2000 [7]. The advantage of this technology over the aforementioned technologies is that TINI works as a stand-alone technology and is available for the KVM/CDC (Kilo Virtual Machine with the Connected Device Configuration). However, in comparison to this Pini is available for the KVM/CLDC (Kilo Virtual Machine with the Connected **Limited** Device Configuration), which has a much more limited functionality than the KVM/CDC.

6 Conclusion and Future Work

This paper demonstrates the possibility of implementing a Jini-like technology that is simple, small and uses RPC instead of RMI. Moreover, spontaneous networking of limited devices may be possible with such a technology. The communicator package provides a simple and efficient way for remote method calls and hence, efficient communication between services and their proxies. In addition, the network traffic can be reduced by submitting only *ServiceDescriptions* instead of full service proxy objects to the clients.

Another measure which would provide more comfort in using Pini, is the development of a compiler, which is able to generate classes like RMI-stubs and skeletons in order to use the object-oriented programming model in the remote context.

Furthermore, we want to make a tool available to connect the Pini technology to the Jini technology. This means, that we will provide a technology, which makes it possible to use Jini services within a Pini community, but also to use Pini services within a Jini community.

References

1. W. Keith Edwards; 1999: **Core Jini**; Prentice Hall PTR; ISBN 0-13-014469-X
2. K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, A. Wollrath; 1999: **The Jini^{TM} Specification**; Addison Wesley; ISBN 0-201-61634-3
3. J. Gosling, B. Joy, G. Steele, G. Bracha; 2000: **The Java^{TM} Language Specification**; Addison-Wesley; ISBN: 0201310082
4. Sun Microsystems: **Java^{TM} Remote Method Invocation (RMI) Specification**
http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html
5. Pabadis Consortium: **http://www.pabadis.org**
6. Stephan Preuß; University of Rostock; 2000: **NetObjects Dynamische Proxy-Architektur für Jini** in "NetObjectDays 2000" work proceedings
7. Alan Kaminsky, Rochester Institute of Technology, 2000: **Running Jini Network Technology in Small Place**, presentation on the "Fifth Jini Community Meeting" http://www.jini.org/jini5/slides/Small_Places/sld001.htm
8. Ch. Nester, M. Philippsen, B. Haumacher; University of Karlsruhe: **Effizients RMI für Java** JIT'99 Java-Informations-Tage 1999; Springer-Verlag ISBN 3-540-66464-5
9. Sun Mircosystems: **Jini^{TM} Technology Surrogate Architecture Specification (Version 0.4)**
10. Sun Microsystems: **Connected Limited Device Configuration, Version 1.0**