CARP@– Managing Dynamic Distributed Jini™ Systems*

Max Breitling, Michael Fahrmair, Chris Salzmann, Maurice Schoenmakers

Technische Universität München
Institut für Informatik
D-80290 München, Germany
{*breitlin|fahrmair|salzmann|schoenma*}*@in.tum.de*

December 16, 1999

Abstract

Jini™ offers the basic technology to develop distributed systems where the participating clients, services and their interactions can adapt dynamically to a changing availability and configuration of the network.[1]

The tool CARP@ (implemented itself as Jini system) is designed to visualize, analyze and control dynamic and distributed Jini systems. The existing reflection mechanisms emerged to be too weak to supply enough information for a suitable management of such a system. Therefore these mechanism had to be extended by realizing a reflective metalevel upon Jini.

This paper describes the tool and its intended usage, and reports the gained experiences together with their implied consequences.

Keywords: *Dynamic Systems, Reflection, Distributed Systems, Jini, Tool Support*

1 Carp@ – a system to observe Jini services

1.1 Modeling Dynamic Distributed Systems in General

As dynamic distributed systems in general as described in (..) might be realized based on different Middlewares like Jini, UPNP or Salutation which might furthermore interact

*This work was supported in part by the Deutsche Forschungsgemeinschaft DFG and the BMW-AG.

[1] Jini and all Jini-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

together in one system, there is an increased need for a middlerware and implementation independent abstract description technique for dynamic distributed systems as such.

From an abstract point of view, DDS consist mainly of service components. It is very important to understand, that these are components on the system level view and may differ from components on the implementation level of view. Service components usually provide thematically grouped functionality on application level, whereas implementation components may realize more than one service with one object. So one can see services as logical or virtual components in contrast to the physical components that actually implement them.

Services of course need to be able to communicate with other services in some way in order to interact in a system kind of way. Communication can be performed in many different ways. In short communication can be targeted or untargeted, synchronous or asynchronous. For targeted communication one might also distinguish between 1:1 and 1:n communication. Targeted communication also requires a service to 'know' its communication partner. This is accomplished by introducing so called *channels*.

As we want to describe distributed systems there is also a need by specifying *locations* that represent a piece of virtual or real hardware on which the services are executed.

Bringing in dynamics we need further elements to describe our system in an abstract implementation independent way. Foremost services can come and go, which introduces no new problems as long as we just want to describe an actual appearance of our system. Next, the implementation of existing services can change transparent during runtime which is also no problem, because we only describe our system implementation independent, by distinguishing between service components and implementation components.

In a dynamic system in principle but also in consequence of new services appearing or services leaving the system during runtime in an unspecified way, also communication links (channels) must be established or cut during runtime, so there should be a possibility to find the right communication partner during runtime. This problem can be solved by adding some kind of type information and selection criteria.

Type information to services is added by the new element *port*. Ports are connectors for channels. Services can have multiple types for accepting communication i.e. messages from other services. These ports are called *inPorts* and further specified according to the kind of communication they can handle

- *interfaceInPorts* for single target synchronous communication
- *eventInPorts* for multi target synchronous communication
- *distributedEventInPorts* for multi target asynchronous communication

As broadcast communication does not require target information there are no channels for broadcasting an therefore no ports respectively.

The channels itself are directed which means that there has to be an equivalent to inPorts on the other end of the channel. This kind of port is called *outPort* and its existence (without a channel) indicates, that the service this kind of port is attached to 'knows' a certain type-information of other services and is able to communicate with them. An outPort with a channel attached to it indicates, that at the moment this service holds a

target reference of another existing service of a given type and is ready to communicate or communicating with this other service.

As inPorts outPorts are named according to the kind of communication they are used for (*interfaceOutPorts*, *eventOutPorts* and *distributedEventOutPorts*).

Services without any inPort and at least one outPort are called *clients* and services can have additional specification information which goes beyond type information which allows to further distinguish between services of the same type and therefore the same external interfaces. This additional service information is called *Properties*. Properties can be everything from additional system structure information, like groups up to detailed functional and non functional specification of a service, e.g. vendor, language, security certificates, extra abilities like printing color and so on.

Actual communication is done by sending *messages* over a channel.

1.2 Modeling Jini Systems on System Level

Of course we want to extract and visualize real systems, in the first step Jini systems, so a concrete mapping between our abstract system model we want to visualize and the implementation of a jini system is needed.

This is quite easy. Services are the public interfaces of an object that is registered in a LUS, clients respectively are objects that hold references to services (at least one) and locations are java VMs.

Channels exists if the attribute reference to a service (outPort) in an object is not null.

InterfaceInPorts are remote interfaces an object registered as a service implements, EventInPorts are implementations of listener interfaces and EventOutPorts a container holding listeners, DistributedEventInPorts are implementations of RemoteEventListener interfaces and DistributedEventOutPorts are containers holding these listeners respectively.

Messages are all method calls that correspond to the interface determined by the inPort.

Properties can be directly mapped to Jini's attributes.

1.3 Getting the System Model of a Jini System by Reflection

Having a mapping between Jini elements and our abstract system model that we want to visualize is not enough, because we are in a dynamic environment, so things can change in not predefined ways which means we have to gather most necessary information at runtime of the system an constantly 'feed' our system model.

1.4 System Architecture

The CARP@-system itself has been designed as a dynamic system using Jini services as its main components, so that it is possible both to manage CARP@ with itself and to extend it during runtime.

The logical architecture of the CARP@ system is shown in Figure 1 with service dependencies from top to bottom. The mobility layer contains services that provide the possibility to start services from remote or to move a running service from one location to another.

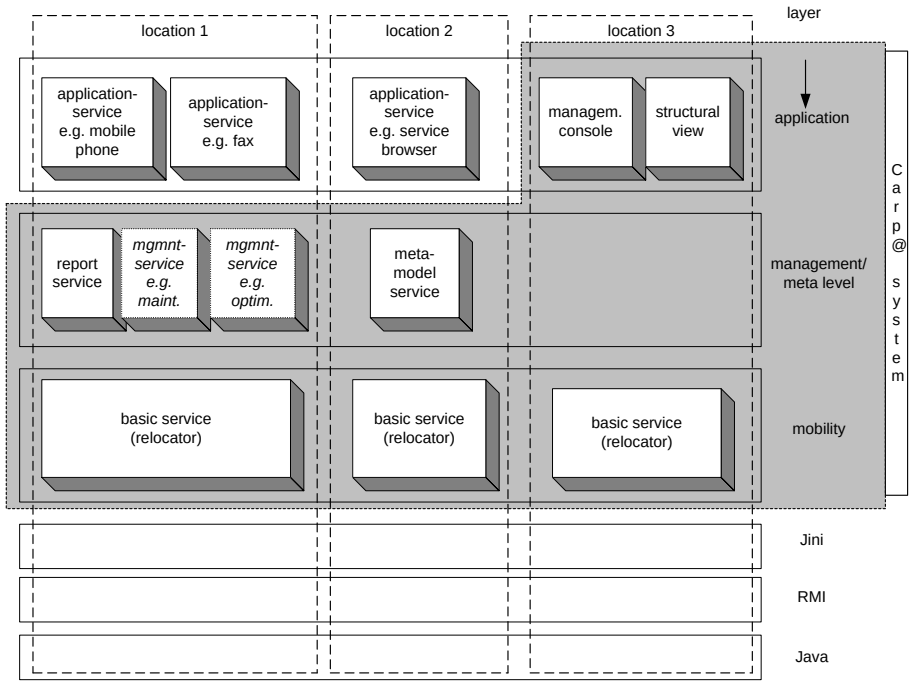

Figure 1: Layers of CARP@ Services

This is an extra layer, because mobility features might be part of Jini in the near future and therefore are not part of the core CARP@ system.

The management layer contains all CARP@ services being involved with gathering, manipulation and storage of information about the observed application system. The CARP@ core system comes with two services assigned to this layer, the report-service and the meta-model-service. The report-service gathers basic information pieces, by querying special meta-level objects, called CARP@-beans (see Figure 2 and Section 1.5), about the observed application ranging from very simple ones, like name and attributes of its services up to complex system structure information like exchanged messages, communication channels or interface ports. All this pieces are stored in the meta-model service that contains a CARP@-internal model of the observed application-system built up from the gathered information. This model might be just displayed by a simple view service assigned to CARP@ s application layer or used to actually manage the observed system by using an extensive console application which might furthermore use specialized management-services to actually control the observed application-system's services e.g. by setting their attributes, changing their names or doing some configuration (Figure 2).

The layer structure shown in Figure 1 does not reflect the actual structure of existing CARP@ components. It is merely a logical view of functional units. The physical implementation could differ from that view. In the actual implementation for example, the report and the meta-model services are grouped together in one service (consisting of two components) due to performance reasons.

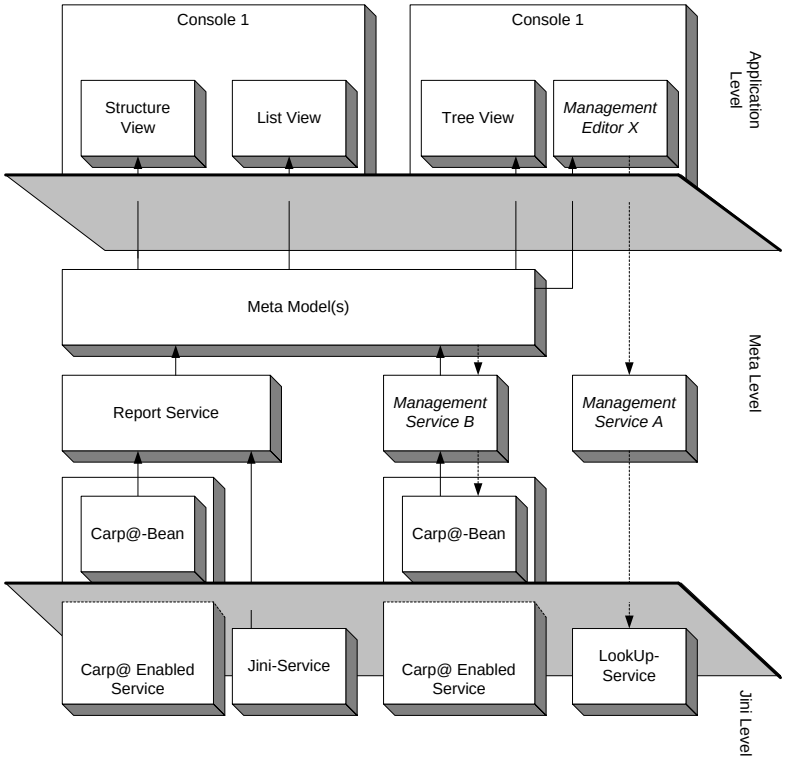

Figure 2: The model collects and propagates data gained by reflection to the user interface views.

1.5 The Meta Level

The basic technique of CARP@ is to find out as much as possible about the system by reflection and other system describing sources. We believe that before a system can be changed at runtime, the first step is to understand and to observe it at runtime. An administrator can then manipulate the system through the model he retrieved by introspection (shown in figure 2). These changes on the meta level then are reflected in the systems runtime behavior.

The model we have chosen to represent the Jini system is not based on classes and references but is an architectural model based on the idea of components and connectors [**?**, **?**]. Figure 3 shows a simplified UML class diagram [**?**] of the meta model that describes this architectural model. A model on an architectural level allows the use of related sets of class instances as single components and hides all the detailed auxiliary classes and objects that are typically used in Java to implement listeners, events and so on. Another advantage is that a connector, here called channel, is a more abstract item then a simple interface reference. So it can describe any kind of communication, like method calls or on distributed events.

Normal Jini-services, clients and standard reflection techniques in Java can not yet deliver the additional information that is required to observe locations, channels and e.g. memory usage. Therefore CARP@ contains a special component model, called CARP@ *beans*. These bean components have to be created by the programmer and extract as much as possible through reflection. To make a Jini service or client fully observable by CARP@,

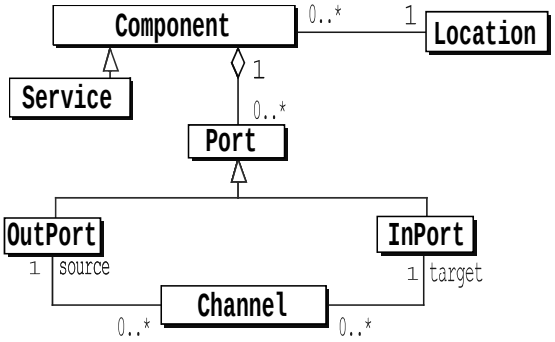

Figure 3: A simplified meta model to describe the architectural elements.

Figure 4: The CARP@ System running

the programmer also has to notify them about changes that can not be detected by standard reflection. So in short some simple programming guidelines have to be fulfilled by the programmer to get full observation possibilities.

Figure 2 shows an example scenario: The Report-Service collects information from both CARP@-bean extended Jini-services and standard Jini-services. This information is stored in the core Meta-Model that contains all basic structure and management information. The CARP@-core-system can be extended by specialized management services that gather additional information being stored either in the core Meta-Model, some additional specialized Meta-Model, e.g. a metric-model (service B), or not at all.

The meta-information can be displayed in different (may be distributed) application-views and can also be changed by editors, either directly (service A) or via reflection of changes in the Meta-Model to the managed system (service B).

1.6 Using CARP@

CARP@ is quite easy to use. In an running Jini-System the user can start the model service. It will investigate and collect data about all services, that are found in all reachable lookup services. The model is constantly updated at runtime.

To see the information about the Jini system that is collected in the model service, the user starts the graphical user interface client and can browse through the system as Figure 4 shows. Multiple clients may exist and are notified constantly about changes while the structure of the Jini-system evolves.

The user interface allows the user to browse through the system to watch all relevant data and to open up different alternative views.

The most intuitive view is the *structure view*. It shows a graphical representation of the collaboration in a Jini system in a ROOM [**?**] like notation. While the Jini system is running, all the views are constantly updated and show the current situation. When a

new Jini service participates in the system, (for example because somebody started it in the network watched by CARP@) it simply pops up as a box. When a Jini service disappears, for example because the service leaves the network, it will be shown grayed and will finally be removed. The graphical layout is automatically performed but can be manually influenced.

Besides the graphical representation in the structure view, CARP@ shows detailed information in various lists. Here the user can see not only the memory consumed by a location, but also conventional Jini information. For example there is group list, that shows the groups, where a service joins at. Jini attributes, that describe the services, are also visualized.

With CARP@ the user can of course not only show all this information, but can also *administrate* the Jini system by adding, changing and removing Jini groups and attributes.

Besides simple administration CARP@ has *management functionality* like starting or stopping Jini components on remote locations, which is very comfortable when more complex test scenarios have to be set up or when the performance with multiple clients has to be tested.

2 Conclusion & Future Work

In this paper we presented CARP@, a management tool for dynamic Jini systems. Since CARP@ needs certain information about the Jini components that are meant to be observed, the reflection mechanisms of Java seemed appropriate. Unfortunately, these mechanisms turned out to be not powerful enough, since e.g. information about the communication links cannot be retrieved sufficiently. Therefore an additional meta level was introduced to make the needed information accessible. A more generalized solution to this problem would be more satifying.

CARP@ is now available in its first beta version [Car]. Future work includes the creation of additional views like message sequence charts [**?**] to visualize the message trace for dedicated parts of a Jini system. Other work will include more specific administrative views for lookup services and java spaces. Management of Jini systems, like migration of services at runtime will be other areas to investigate.

However, making a Jini service or client fully observable by inserting code at the source code level is too retrictive. Currently we are working on an integration of a class file transformer that instruments the code at runtime on a bytecode level. Tools like JOIE [**?**] will be used for this. The advantage is that also components where no source code is available can be observed completly. The byte code transformation is done again with reflective techniques based on a the meta information contained in the class file. Because the code must be changed before it is loaded, normal Java reflection can not be used.

Acknowledgments We would like to thank the whole CARP@ team for a lot of overtime work.

References

[Car] CARP@ Homepage. `http://www4.in.tum.de/~carpat/`.

[Cor] CORBA Homepage. `http://www.omg.org/corba/`.

[EE98] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[OMG92] OMG. Object management architecture guide – revision 2.0, 1992.