CARP@– Managing Dynamic Jini™ Systems*

Michael Fahrmair, Chris Salzmann, Maurice Schoenmakers

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

November 15, 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]
Jini can be seen as an dynamic middleware layer on top of the distribution middleware. The tool CARP@ (say *Carpet*) is designed to visualize, analyze and control dynamic and distributed Jini systems. Unlike usual Jini browsers CARP@ allows the observation of services *and* clients, their interconnections and the messages exchanged between them. The tool extracts an *architectural component model* based on *components*, *ports* and *connectors*. Based on this model the structure within the dynamic middleware layer should be changeable at runtime without changing the code. This paper describes the tool, its intended usage and the work in progress.

Keywords: *Dynamic Systems, Dynamic Architecture, Middleware, Distributed Systems, Jini, Tool Support*

1 Introduction

The areas and complexity of applications where computer systems are used is growing constantly. More and more devices are controlled by computers, nearly all data is meanwhile processed by computers, and with a spreading Internet more and more computers are interconnected. This leads to a demand to benefit of the emerging new possibilities. The main problem is not to be seen as a hardware problem, but as a software problem: The growing complexity makes it difficult to develop correct programs that perform the intended tasks. The problems are mainly caused by the following two characteristics of the systems to be built:

- The systems are *distributed*: A system consists of multiple active participants that are interacting together to perform a certain task. This interaction is achieved by communication between them.
- The systems are *dynamic*: The architecture, i.e. the presence of the components, their arrangement, their implementations and their interconnections, but also the roles they take (as e.g. *server, service, client*) are changing during the runtime of the system. Due to the need for a high availability of systems it is often no longer possible to stop or interrupt them for reconfiguration.

To tackle these problems, suitable programming paradigms, languages and tools are needed. Middleware technologies such as CORBA [OMG92, Cor] and DCOM [EE98] are first approaches in this direction. Jini is the first product that claims to solve the mentioned problems concerning dynamic configuration. It offers interfaces and mechanisms for components to announce their own abilities, to look for services of

*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.

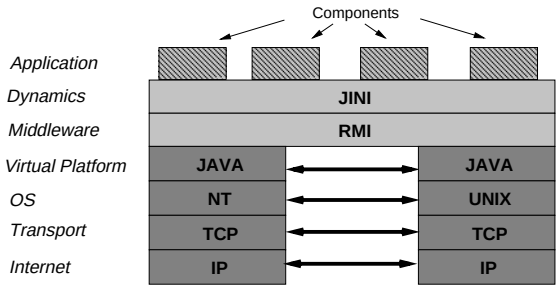

Figure 1: The layers of a Jini System

other components and use these in a dynamic network of interacting components. Figure 1 shows how Jini can be seen as a dynamic middleware layer that relies on the more classic distribution technology RMI.

Using Jini, it is now possible to develop dynamic, distributed systems in a more or less comfortable way. But, a lot of the interaction going on in a Jini system is hidden. On the one hand, it is of course desirable to keep things transparent if a developer wants to focus on the intended application he is creating. But, on the other hand, it should be possible to watch and analyze these hidden interactions if something goes wrong (for debugging) or is inefficient.

The tool CARP@ [Car] was developed to watch and visualize a network consisting of several Jini components, together with the possibility to manage such a Jini system, both at runtime. To get a manageable overall view of the system CARP@ extracts an architectural model of a Jini system at runtime. This model consists of *components*, *ports* and *connectors*. Components can be Jini clients and services. Ports are points where services are provided or used and connectors represent different kinds of communication like method calls or events [SG96, SS99]. The idea is to observe and manipulate a dynamic distributed system in terms of the architectural model. CARP@ will then transform these changes at runtime to changes in the Jini system. (see Figure 2).

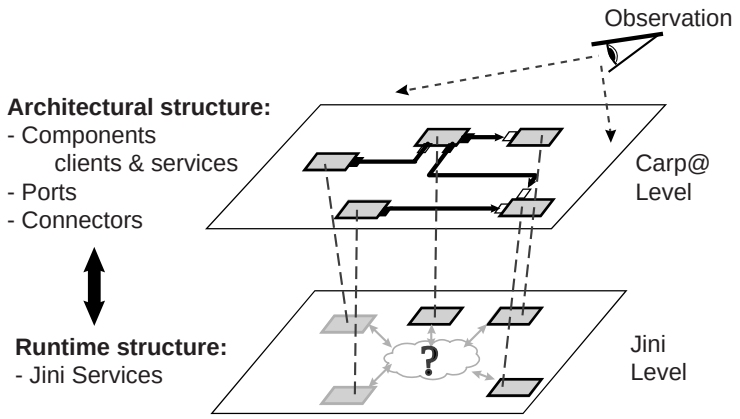

Figure 2: Observing a Jini System via its CARP@ meta level.

The project was performed until now in a one semester period with a dozen of graduate students and the authors. CARP@ itself is implemented using Java and Jini to gain additional knowledge and experience using these technologies.

In this paper, we present in Section 2 the tool as far as it exists today. We conclude with our future work in Section 3 and also summarize the results in this last section.

2 Observing and Managing Jini Systems with CARP@

CARP@ is a tool to observe, administrate and manage a dynamic network of Jini components with all their communication relationships at runtime.

In a dynamic ad hoc networking environment, the actual architecture evolves during runtime. Decisions like choosing an implementation for a component or establishing a communication structure are not taken at design time but at runtime. Therefore in our opinion there exists an increased need to extract an architecture description at runtime. This description then can serve as a base to decide about the effect of changes.

Thus CARP@ goes beyond showing simple Jini-services like other browsers do and shows additional important information that is not available otherwise but is needed to understand the interaction in a Jini system:

- *Clients and Services*
  CARP@ shows besides the services also the *clients* of these Jini services and how these components communicate with each other through so called *channels*.
- *Messages*
  CARP@ enables the developer to trace the methods calls, together with their arguments, as *messages*, that are sent between services and clients for each channel in the system.
- *Provided and Required Interfaces*
  Services can provide multiple interfaces and clients may require multiple services. CARP@ shows not only these *provided* service interfaces, but also the *required interfaces* as *ports* for each component.
- *Locations*
  An information also not available in general Jini systems is the *location* of a service or a client. This is for example important if you want to know where misbehaving clients are located or you want to see where a performance bottleneck of a system occurs.

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 [SG96, SS99]. 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.

CARP@ is divided up in several components. Each observable Jini client or service should contain a so called CARP@-bean. Each CARP@-bean maintains the data concerning this single component. A *model service* looks for CARP@-beans and creates a consistent architectural model of all nearby clients and services.

To see the information about the Jini system that is collected by the model service, the user starts a graphical user interface client and can browse through the system as Figure 3 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 [SGW94] 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, like groups and service attributes.

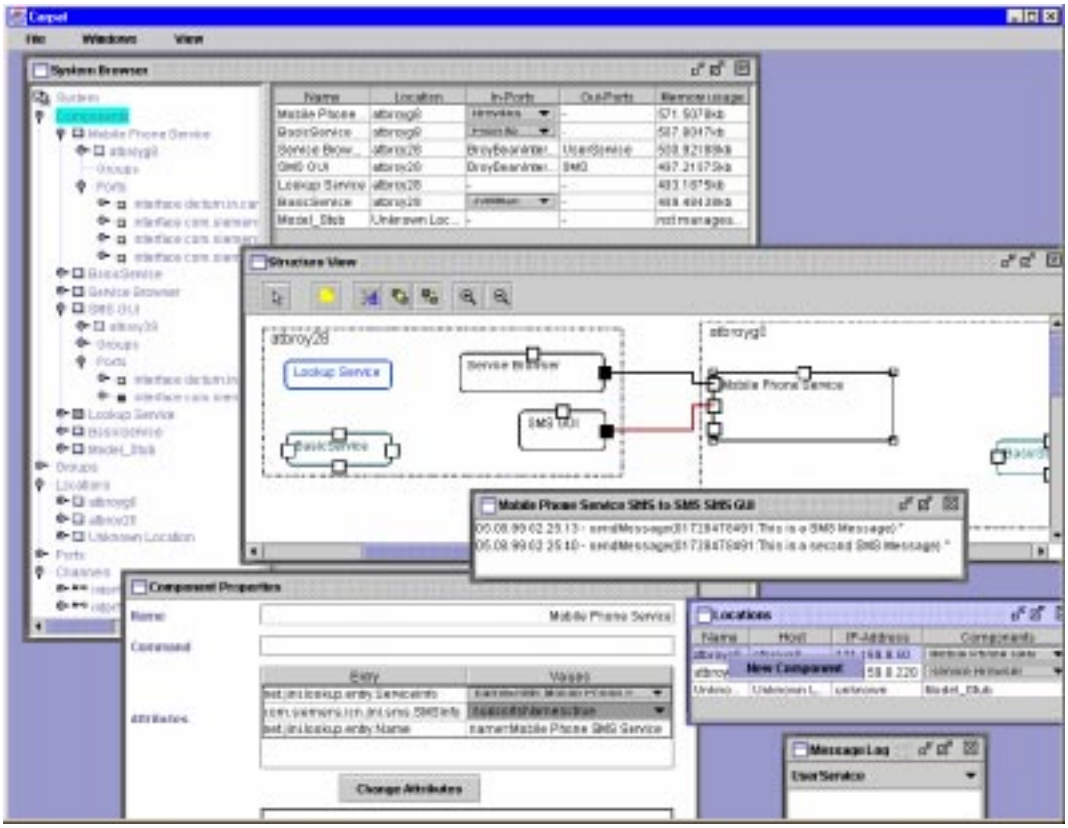

Figure 3: The CARP@ System running

With CARP@ the user 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.

To make a Jini service or client fully observable by CARP@ the programmer has to use CARP@ bean components. Because some changes can not be detected by standard reflection, some simple programming guidelines have yet to be fulfilled by the programmer to get full observation possibilities.

3 Conclusion & Future Work

In this paper we presented CARP@, a management tool for dynamic Jini systems. We used CARP@ in a Jini project and the observation abilities where found very useful.

It is still somewhat unclear how an exact mapping between the object model and the architectural model should be defined. So where should be the boundary around a collection of small objects like a service objects and several different listener objects to view them as one single component or connector.

Another open issue is the definition of system situations where it is "save" to modify the configuration of a system. How should constraints be defined and enforced that define that it is save to perform a change? Examples of the configuration changes would be changing the location of a service, exchanging a service implementation or changing the wiring between components on the fly at runtime. More generalized solutions to these problem would be more satisfying.

CARP@ is now available in its first beta version [Car]. Future work includes the creation of additional views like message sequence charts [Krü] 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 restrictive. Currently we are working on an integration of a class file transformer that instruments the code at runtime on a byte code level. Tools like JOIE [CCK98] will be used for this. The advantage is that also components where no source code is available can be observed completely. The byte code transformation is done 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. But before the code instrumentation can be achieved automatically the mapping mentioned above must be defined.

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/`.

[CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with joie. In *Proceedings of USENIX Annual technical Symposium 98*, 1998.

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

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

[Krü] Ingolf Krüger. Towards the methodical usage of message sequence charts. In Katharina Spies and Bernhard Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme. FBT99.* Herbert Utz Verlag, 1999.

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

[SG96] Mary Shaw and David Garlan. *Software Architecture – Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real–Time Object Oriented Modeling.* Wiley & Sons, 1994.

[SS99] Chris Salzmann and Maurice Schoenmakers. Dynamics and mobility in software architecture. In Jan Bosch, editor, *Proceedings of NOSA 99 – Second Nordic Workshop on Software Architecture*, 1999.