Using Jini and JavaSpaces with Ericsson NorARC's technologies for service creation

- A new way of adapting to dynamic environments -

Masters thesis
in
Information and Communication technology

Fritjof Boger Engelhardtsen
Tommy Gagnes

Grimstad, May 2002

Summary

Ericsson NorARC (Norwegian Applied Research Center) is developing architectures and frameworks for advanced telecom and Internet systems and services. The current technologies are JavaFrame, ActorFrame and ServiceFrame, which enable rapid development and system dependability.

Jini is a technology for making dynamic, robust and self-healing distributed systems. It can be used with JavaSpaces technology, which is a distributed shared memory technology that provides loosely coupled communication in time, space and destination.

In this thesis, we evaluate if Jini, JavaSpaces and frameworks building on these technologies can be used to distribute Ericsson NorARC's technologies in dynamic network-centric environments.

We have evaluated the relatively immature frameworks Rio, SI and Openwings. All the frameworks have interesting features that could prove to be valuable, but as it is today, Openwings stands out as the most complete and useful framework. It also has properties that match well with JavaFrame concepts.

Based on Jini and JavaSpaces, we have developed several conceptual ideas. This has been done to show how these technologies could be used together with the JavaFrame, ActorFrame and ServiceFrame concepts of asynchronously communicating state machines. Several interesting ideas have evolved when having taken an in-depth look at Jini and JavaSpaces, for instance a conceptual space-based architecture has been developed. In this architecture, we introduce new concepts like dynamic role downloading, a Role repository and role trading.

A prototype framework called SpaceFrame has been developed to simplify distribution of system components. Using SpaceFrame will also help systems adapt to change, in both system demands and network topology. An example application has been developed by using SpaceFrame, demonstrating loosely coupled communication and downloading of roles on the fly. SpaceFrame shows that integration of Jini, JavaSpaces and JavaFrame-based technologies is possible, much due to the fact that Jini's distributed event model can be used for asynchronous communication through a JavaSpace.

We have found that Both Jini and JavaSpaces are technologies that have something to offer when distributing Ericsson NorARC's service creation technologies in dynamic context-sensitive environments. The most elegant model is probably obtained by combining Jini and JavaSpaces together as we have done with the SpaceFrame prototype. By using this approach, total asynchronous and uncoupled communication can be achieved between different JVMs.

Preface

This thesis was written for Ericsson NorARC (Norwegian Applied Research Center) and is part of the Norwegian master equivalent "siv.ing." degree. The work has been carried out in the period between November 2001 and May 2002. We have spent most of the time working at Ericsson NorARC's laboratory in Asker, but the report has been written at the Agder University College in Grimstad. The thesis is part of the "Mobile Student" R&D program, in which both Ericsson and Agder University College are involved.

We would like to thank our supervisors, Jan P. Nytun at Agder University College and Knut Eilif Husa at Ericsson NorARC, for valuable help and inspiration. We would also like to thank Øystein Haugen, Geir Melby and Stein Bergsmark at Ericsson for their valuable comments during our work.

Grimstad, May 2002

Fritjof Boger Engelhardtsen and Tommy Gagnes

Table of contents

List of figures

List of tables

1 Introduction

1.1 Background

The number of Internet users has faced a massive growth during the last decade, as has the number of mobile phone users. There is reason to believe that these numbers will continue to grow during the next decade, possibly at an even faster rate, as we face the roll-out of the 3rd generation mobile systems.

As mobile terminals become more powerful in terms of processing power and memory, the Internet is expected to be accessed by mobile phones and other devices to a much greater extent than it is today. This will be done through a variety of different protocols and standards with the Internet Protocol as a lowest common denominator. As the use of the IPv6 address space increases, it is highly plausible that we in addition to PCs also will see many other appliances, including mobile and embedded devices, connecting to the Internet.

Some are already talking about 4th generation systems, not meaning new bearer technologies, but convergence between different existing bearer technologies, such as Wireless LANs and UMTS. Mobility and "always-on" will be key concepts, with a rapidly changing environment where services and clients come and go all the time.

With Internet becoming available everywhere, telecom operators, Internet Service Providers and third-party vendors will need to be able to quickly develop new services for their customers. The competition will get harder as there is only one huge network, the Internet, consisting of many smaller interconnected networks. With all these ubiquitous devices and one, big network, it is fairly reasonable to say that much of the competition will be focusing around services. For service providers to be competitive, short time-to-market for complex new services will be essential.

This chapter gives an overview of the technologies that we use as a foundation for our work. We start with a brief introduction to Ericsson NorARC's service creation technologies, before presenting Jini and JavaSpaces and related systems. We continue with a presentation of the thesis definition and explain where our work fits in with these technologies, before giving an outline of the rest of the report.

1.2 Ericsson NorARC's service creation environment

To deal with the above mentioned complex services, Ericsson NorARC (Norwegian Applied Research Center) is developing the ServiceFrame architecture. ServiceFrame [1] is a service execution framework that contains specific functionality for advanced telecommunication and Internet services. ServiceFrame is layered on top of the more general ActorFrame and JavaFrame frameworks.

JavaFrame [2] is a Modeling Development Kit (MDK) which targets large, complex real-time systems written in the Java programming language. These systems are modeled as active objects (state machines and composites) that interact asynchronously through message passing. To model the system behavior, the emerging UML 2.0 standard is used. In order to provide independence, mediators represent the interaction interfaces between active objects.

ActorFrame [3] builds on JavaFrame, and is based on the concepts of actors and roles. Actors can play different roles, and the relations between actors are organized into a nested hierarchical tree structure.

Currently, only JavaFrame has been released, whereas ActorFrame and ServiceFrame still are under development. For distribution of these technologies in dynamic environments, no ideal solution has yet been developed.

1.3 Jini technology and related frameworks

Jini is an object-oriented framework for building scalable, robust, distributed systems (using Java) [4]. Since Ericsson NorARC's technologies all build on the Java programming language, it will be interesting to see what Jini can offer in combination with these.

Jini [5] enables computers to find each other and use each other's (software) services on a network without any prior information about each other, only about the service wanted. This is known as a service discovery mechanism, and in Jini, service discovery is done by using proxy objects, mobile code, IP multicast and a lookup service (LUS). Several new service discovery-like technologies are available, including Microsoft's Universal Plug and Play [6] Web Services [7] and JXTA [8]. The Universal Plug and Play architecture mainly targets home networks, whereas Web Services in most cases will need compile-time knowledge of the protocol used. The JXTA project is developing a peer-to-peer framework. As opposed to Jini, these technologies are XML-based, not truly object-oriented, and thus cannot pass behavior along with the data that is sent over the network.

The Jini framework consists of several components, including different protocols, classes, interfaces and services. Jini builds on Sun's Remote Method Invocation (RMI) [9], which passes serialized objects, possibly with behavior, over the network as explained in [10]. It therefore differs from traditional approaches to the distributed computing problem, such as Remote Procedure Calls (RPC), CORBA (Common Object Request Broker Architecture), and the new XML-RPC standard [11].

Using Jini means that it is not necessary for all parts involved in a distributed system to actually know which protocol to use at compile time. Traditional technologies such as

RPC and CORBA require that each side of a connection have prior knowledge of the protocol used. With Jini, the service itself can provide the implementation of the protocol that must be used by the client. Actually, this might be any kind of protocol, including the CORBA IIOP protocol.

The Rio Architecture [12] is a basically a Quality of Service (QoS) component model for Jini services. There are several similarities between the Rio architecture and the Enterprise Java Beans component model. The QoS concept is introduced to deploy service components on the most capable computer.

Openwings [13] is an architecture that is being developed by Sun and General Dynamics Decision Systems (formerly Motorola IISG). It is described in [13] as «…a service-oriented architecture where components spontaneously publish and discover services, independent of transport protocols and deployment environments». Openwings aims to be an abstraction on top of various service discovery mechanisms, including Jini. The discovery mechanisms function as plug-ins in the framework. Openwings utilizes a container, in which services can be deployed. The container provides a middleware layer, so that the developer can focus on domain-specific issues.

1.4 Space-based systems and architectures

JavaSpaces is a specification [14] of a Java-technology that is closely related to Jini. It is a Distributed Shared Memory (DSM) technology that can be used as a Jini service, and builds on the Linda Tuple Spaces concept developed by David Gelernter at the Yale University, first published in 1985. This space concept opens up many possibilities. The use of space-based technology can provide three forms of uncoupled communication, namely uncoupling in space, destination and time [15]. A sender does not have to be aware of a receiver's location and address, neither do the two (sender and receiver) have to exist at the same time. But still, they can exchange messages through the space, which will store them until they are read.

Sun's JavaSpaces utilizes Java RMI to write and read objects called entries to a JavaSpace. Entries are stored in serialized form in the space, and are de-serialized when read by a "client" of the space.

There are still only a few Java-based products that build on the space paradigm. Sun's reference implementation, Outrigger, is included with the Jini software. According to [16], Outrigger can hold between 10.000 and 100.000 entries.

GigaSpaces™ is an example of a platform that is compatible with the JavaSpaces specification. This is a commercial product that has the following vision: «Applications should be able to communicate and share information anywhere, anytime and without

prior knowledge of each other» [17]. The GigaSpaces product integrates with Web Services (UDDI, WSDL) [7] and SOAP [18] and more spaces can be clustered.

Another implementation of the JavaSpaces specification is IntaSpaces [19] from IntaMission, which is said to be an "evolvable space". IntaSpaces also integrates with Java 2 Enterprise Edition [20] and can hold 2 million entries per space, according to [16].

Tspaces [21] is another technology that builds on the Linda system. The Tspaces product is not building on the JavaSpaces specification, but is written in Java. It is being marketed by IBM as a product that «…enables communication between applications and devices in a network of heterogeneous computers and operating systems».

The Spanish Inquisition architecture or Scalable infrastructure [22] is being developed by Cisco Systems, and is an architecture built on top of the Jini and JavaSpaces technologies. According to [22], it started out as «…a project that would potentially allow an infinite number of devices/users to attach to a communications architecture and fulfill its mission under severe time restrictions in an almost real-time environment».

The technologies mentioned above have chosen the tuple space principles to create products that:

- Provide the uncoupling in time, space and destination as mentioned above
- Can be used in heterogeneous environments
- Are scalable and reliable

We found this inspiring when we began to look at combining Ericsson NorARC's technologies for service creation and the Jini and JavaSpaces architectures.

1.5 Thesis definition

We wanted to see if using Jini and JavaSpaces could be beneficial when distributing Ericsson NorARC's technologies in dynamic environments. The final thesis definition therefore was formulated like this:

«The students will evaluate the Jini and JavaSpaces architecture and programming model with respect to JavaFrame's concept of asynchronous communicating state machines and mediators. They will also evaluate if related frameworks such as Openwings, SI and Rio can be used as middleware for Ericsson NorARC's technologies for service creation.

If possible, a prototype will be made, demonstrating some of the concepts mentioned above».

We have not found it necessary to change the thesis definition during our working period.

The initial thesis title formulated like this:

«Using Jini with Ericsson NorARC's technologies for service creation».

As the thesis definition suggests, an evaluation of both Jini and JavaSpaces will be done. We have therefore, in agreement with Jan P. Nytun, our supervisor, updated the thesis title to reflect this. The final thesis title is therefore formulated like:

«Using Jini and JavaSpaces with Ericsson NorARC's technologies for service creation».

1.6 Our work

Currently, the NorARC technologies for service creation in many cases are not ready to be used in dynamic environments. In these environments, clients and services come and go all the time and system components may dynamically be added and removed. The objective of our thesis therefore is to look at how NorARC's service creation technologies could be adapted to this kind of environment. We will therefore have to look at how we dynamically could distribute JavaFrame-based system components, and how to communicate asynchronously between computers in such an environment. Jini and JavaSpaces are the technologies that we have chosen to try to achieve this, because they are designed to be used in such environments.

The questions to be answered therefore are:

- Can Jini, JavaSpaces or frameworks building on these technologies be used to distribute Ericsson NorARC's service creation technologies?
- If yes, can this be done in an efficient and sensible way, without conflicting with JavaFrame's principles?

To answer these questions we first evaluate how existing frameworks and architectures building on Jini and JavaSpaces might be used with Ericsson NorARC's service creation environments. There are several research projects that build on Jini and JavaSpaces, including the Rio [12], Openwings [13] and SI [22] architectures. These projects aim at making it easier to develop and deploy systems bases on Jini and JavaSpaces. It will therefore be interesting to see if these architectures might have any valuable concepts to offer or may be used as middleware for JavaFrame-based systems.

We also describe different conceptual solutions that we have developed in order to introduce Jini and JavaSpaces in a JavaFrame world, including a conceptual space-based architecture. The interesting question is if this can be done in an elegant and beneficial way that makes sense. It is not enough to integrate the technologies if no value-adding benefits can be drawn from it.

A prototype of our conceptual space-based architecture has also been developed. The prototype, which is called SpaceFrame, builds on JavaFrame, JavaSpaces and Jini, and demonstrates key concepts from the conceptual architecture.

We have also developed an example application to show how SpaceFrame can be used. Such an example application can verify that our space-based architecture is possible to realize.

Figure 1.1 shows how SpaceFrame builds on JavaFrame, JavaSpaces and Jini. Ultimately, ActorFrame should be able to integrate and take advantage of SpaceFrame but in this project we have focused mostly on the "layers below" ActorFrame (Figure 1.1).

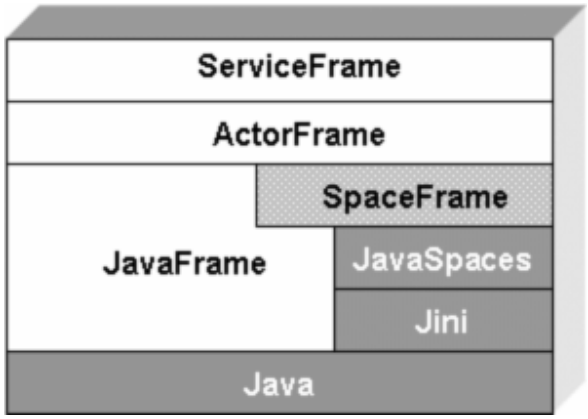

Figure 1.1 Framework stack

1.7 Report outline

In the following chapters we will take a closer look at some of the technologies mentioned above. In chapter 2, we will present some of the characteristics of the NorARC technologies before describing the Jini and JavaSpaces technologies in chapter 3. The frameworks mentioned above; Rio, Openwings and SI are explained briefly in chapter 4.

In chapters 5 to 10, we will present our work, starting with an evaluation of Rio, Openwings and SI in chapter 5 with respect to JavaFrame concepts. After an introduction to our conceptual solutions in chapter 6, thoughts on how Jini can be used together with Ericsson NorARC's technologies are presented in chapter 7. A conceptual space-based architecture and the SpaceFrame prototype are presented in chapters 8 and 9, respectively. We also present an example application developed with SpaceFrame. This is done in chapter 10. In chapter 11, we discuss our results before a conclusion is given in chapter 12.

2 Ericsson's service creation architectures

2.1 Introduction

JavaFrame, ActorFrame and ServiceFrame are three frameworks that are being developed by Ericsson NorARC. They all build on the Java programming language. All the three frameworks are layered on top of each other (shown in Figure 2.1). ServiceFrame is dependent on ActorFrame and JavaFrame while ActorFrame is dependent of JavaFrame. How much of the framework stack that is used is up to the designer to decide. The frameworks are based on asynchronously communicating active objects, which communicate by passing messages through mediators. An example of an active object is a state machine. According to [1], «Communicating state machines has proven to be an excellent conceptual abstraction in the domain of communication and real-time systems».

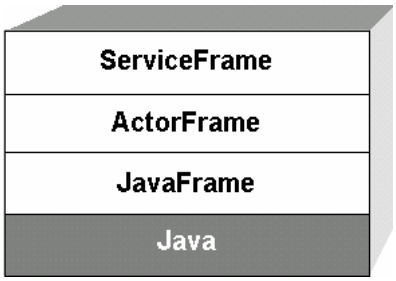

Figure 2.1 Ericsson's service creation frameworks

2.2 JavaFrame, a foundation for complex Java based systems

At the bottom layer is JavaFrame MDK, which «…aims to improve the dependability of large complex real-time systems implemented in Java» [2]. According to [2], «…using JavaFrame will also make the resulting system more maintainable through the independence of system parts». JavaFrame includes the notions of mediators, state machines, composites and composite states. A JavaFrame system can be modeled by using the emerging UML 2.0 standard. There is a one-to-one relationship between the Java code and the model of the system. This implies that a change in the code automatically should lead to a change in the model, and vice versa.

2.2.1 Active objects

In JavaFrame, an «ActiveObject is an abstract class representing the notion of something that acts on its own, sending and receiving messages through its associated mediators» [2]. There are two kinds of active objects, Composites and State Machines. Scalability and composition is achieved by allowing Composites to contain nested structures of active objects. The Addressable interface represents addressable entities that can emit and consume messages. Figure 2.2 (taken from [23]) shows a conceptual UML diagram of JavaFrame's active objects. Asynchronous operation is achieved by giving each State

Machine a buffer for incoming messages, thereby enabling different system parts to work at their own speed. Each JavaFrame application has one or more Schedulers, which is again associated with a thread, managing the message queues on the state machines.

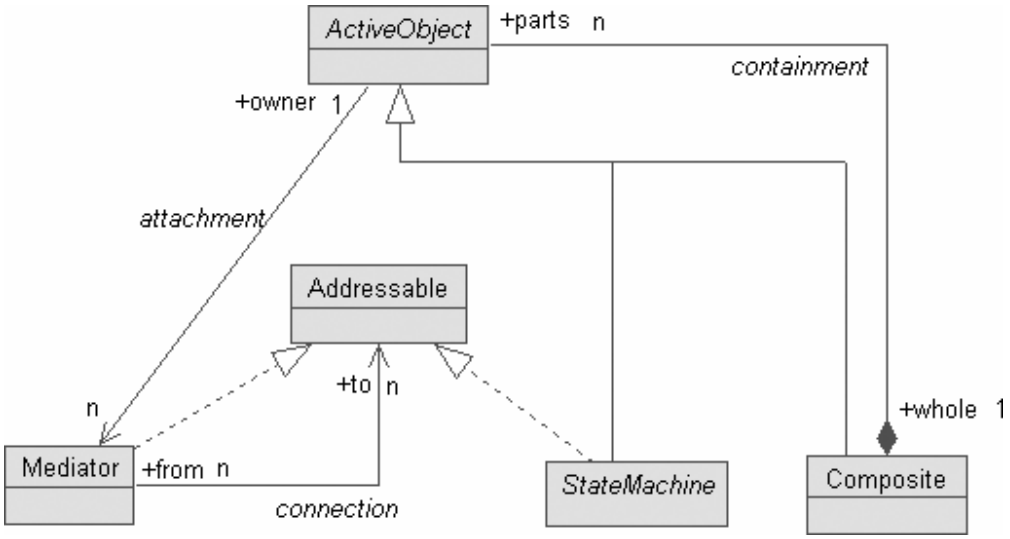

Figure 2.2 JavaFrame's ActiveObject concept

2.2.2 Composite states

Composition can also be achieved with Composite States. Composite States are simply states containing inner states. Somewhat like composites that contain nested structures of active objects, Composite States can contain inner nested structures of states and Composite States. This is shown in Figure 2.3.

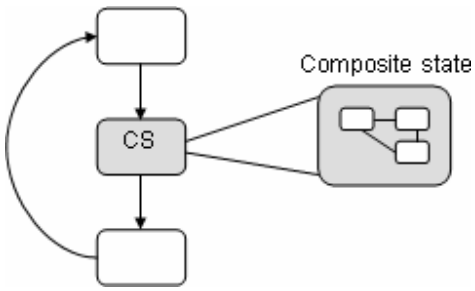

Figure 2.3 Composite states

2.2.3 Mediators

Mediators are used to achieve independence between active objects. All communication between active objects passes through mediators. The mediator concept separates low-level communication, and thereby possibly platform-specific needs, from the program logic in the state machines. Several types of mediators, including simple mediators, multicast mediators, router mediators and edge mediators, exist. Simple mediators are merely used for message forwarding while edge mediators are used to communicate with other system instances. Input mediators are colored gray while output mediators

are white, as shown in Figure 2.4. For more details about JavaFrame, please refer to [2] and [23].

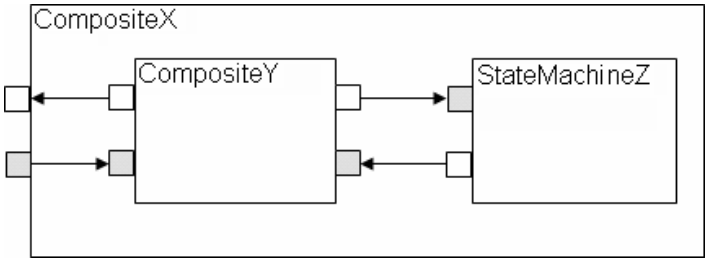

Figure 2.4 Mediators

2.3 ActorFrame

ActorFrame [3] lies above JavaFrame and is a relatively new development at the time of writing. ActorFrame is ServiceFrame's service execution framework without ServiceFrame's internet and telecom domain-specific functionality. Complexity and composition is handled in ActorFrame with the actor-abstraction. Actors represent entities that can execute different behaviors, that is, play different roles. The notion of a play means a context that may have several members that need some sort of connectivity with each other. This connectivity is set up by the Role-request protocol which is used to determine if an actor can play a requested role. An actor can contain an inner nested structure of child-actors. ActorFrame relies on deployment descriptors that are used to set up the initial structure of actors and set restrictions and properties for how these actors are created and interact during runtime.

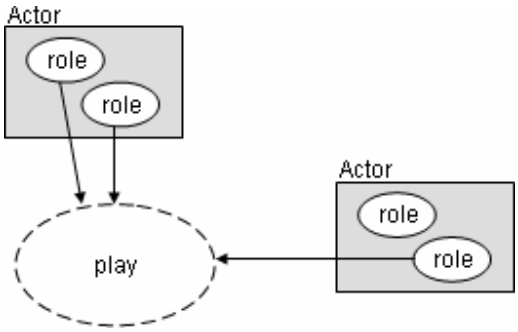

Figure 2.5 Actors, roles, and a play

2.4 Service creation in ServiceFrame

At the top layer lies the ServiceFrame framework, which «…is a service execution framework for advanced, hybrid and personalized services» [1]. Core concept in ServiceFrame are mirroring the environment into the system and using a service-centered approach. By separating services from underlying system dependencies, one may avoid creating different versions of the same system over and over again. New services are rapidly created by specializing an already existing set of domain objects. The notion of a service role is introduced and complete service roles will most often be represented by composite states. ServiceFrame contains Internet and telecom specific domain objects like UserManagers, TerminalManagers and ServiceManagers etc.

2.5 Summary

We have seen that Ericsson NorARC's technologies for service creation represent several concepts to improve efficiency and modeling of services. For more information about JavaFrame, ActorFrame and ServiceFrame, [2], [23], [3] and [1] are included in Appendix A.

In the following chapter we will look more deeply into two other technologies that can be used to develop services, namely Jini and JavaSpaces.

3 Jini and JavaSpaces

3.1 Introduction

In this chapter we will look more deeply into the Jini and JavaSpaces technology basics. Since many of the mechanisms in Jini form the background for our conceptual solutions and our space–based prototype, a basic understanding of Jini is needed. The tuple space paradigm and a description of the JavaSpaces technology are also covered in this chapter.

3.2 Jini technology basics

3.2.1 Introduction

Jini is a specification [5] that has been developed by Sun Microsystems and builds on the Java programming language. It specifies a self-healing, service-oriented distributed architecture for dynamic environments. Jini was first released in 1999, promoted as a device plug-and-play technology. Recently, Sun announced that more emphasis should be placed on using Jini for software services. This is nothing new, but it may help explain why Jini has not yet become as widely used as first assumed. In this thesis, we focus mainly on using Jini for software services. However, in the example application presented in chapter 9, we do use Jini for connecting a device to a software system.

3.2.2 Jini services, groups and federations

Jini is based on the concept of a *service*. A service is a logical entity that represents a piece of work that can be done on behalf of a client. The notion of a service helps abstracting away from the concrete computer where this work may be done. Rather than knowing the name of a server that may perform various types of work, one could simply find the concrete service by asking specifically for it. A service can again consist of other services, thus acting as a client when communicating with these. Services can be organized into *groups*, which specify network boundaries for services. On demand, services may also create a *federation*, which is a temporary union of Jini clients and services [4]. Federations can also be linked together, which gives great flexibility. Figure 3.1 shows three groups. The client, A, requests a service from B, which again is part of a federation with C and D. B therefore acts both as a service and a client in this case.

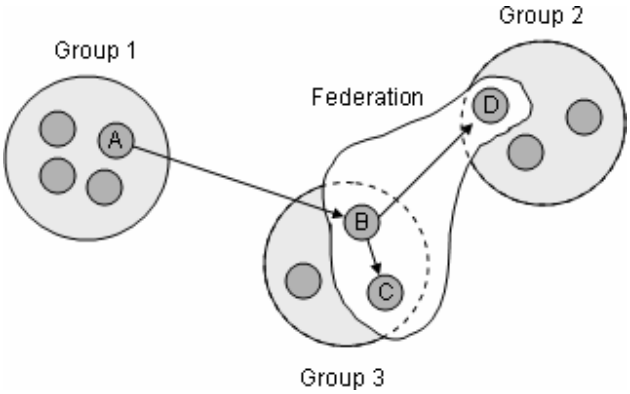

Figure 3.1 Groups and federations

3.2.3 Finding services through discovery and lookup

The Lookup Service (LUS) is a Jini service that helps clients to find services. A Jini system can have multiple lookup services, which help reduce the load on each and provide replicated information. When bootstrapping a Jini network, a Discovery Protocol is used to enable both services and clients to obtain a reference to the Lookup Service. This is a low-level protocol that relies on IP multicast. When a service joins or leaves a federation, it must adhere to a set of requirements to behave correctly and thereby ensure a stable system. These requirements form the Join Protocol. Fuller descriptions of these protocols can be found in [4] and [5].

When a service wants to publish itself, it must register a proxy object with the Lookup Service. A client wishing to make use of the service initially does not know about its location or existence. Therefore, it must query the Lookup Service for a service that can fulfill its wishes. If such a service is found, the client downloads a copy of the proxy object. Now the client uses the proxy object to contact the service directly, with the protocol embedded in the proxy object. This process is shown in Figure 3.2.

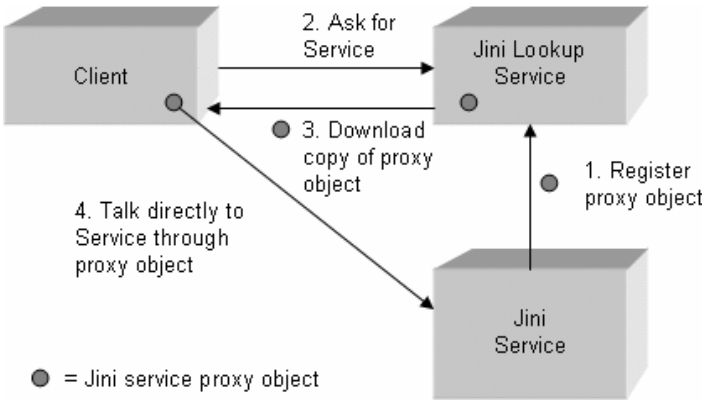

Figure 3.2 Download and use of proxy object

The queries done on the Lookup Service are based on service items that describe the services. Service items are stored in the Lookup Service and consist of the fields shown in Table 3.1, which is taken from [24]. The AttributeSets field is optional.

Table 3.1 Elements of a service item

Field	Description
ServiceID	The identifier of the service associated with this item
Service	A (possibly remote) reference to the object implementing the service
AttributeSets	A set of tuples describing the service

3.2.4 Leasing

To make Jini self-healing, leases are utilized. Nearly every registration or resource must be leased, that is, it must be confirmed that the registered resource is alive or that there is still interest in a resource. If the lease is not renewed before its expiration, the resource or registration becomes unavailable. This provides a form of distributed garbage collection, where only the healthy resources continue to be published.

3.2.5 The Jini Distributed Event model

In the Jini specification, a distributed event model is also specified. This is the distributed equivalent of the traditional Java event model, where an event listener must register a reference to itself with the event source. An event object is fired when the specified event occurs. In the distributed event model, a remote event object is passed to the remote event listener via the network by using RMI. This is shown in Figure 3.3, which is taken from [25].

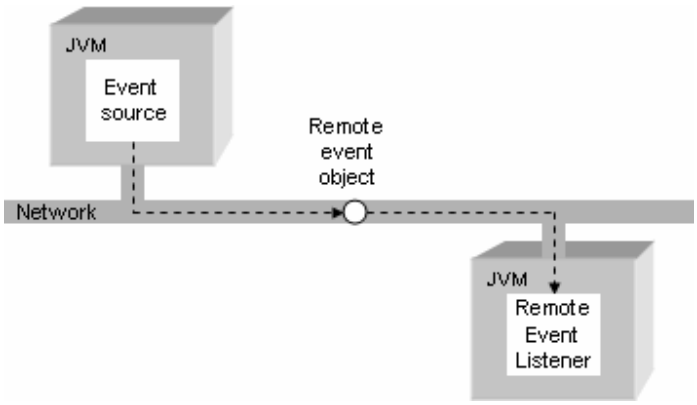

Figure 3.3 The distributed event model

3.2.6 The Jini Distributed Transaction model

Support for distributed transactions is also specified in the Jini specification. This makes it possible to control whether an action can be completed by all the involved parties, or

not. Appropriate action can be taken if any of these cases occur. The distributed transaction specification provides a framework that services can use to give transactions ACID properties [4]. ACID stands for Atomicity, Consistency, Isolation and Durability, which are important issues both in transaction theory and practice. The Jini distributed transaction model is based on the well-known two-phase commit protocol. For further information about ACID properties and the two-phase commit protocol, please refer to [10].

3.2.7 Instantiating new services with the Jini Start-library

It is worth mentioning that in Sun's reference implementation of Jini, a package called `com.sun.jini.start` is included. This package includes utilities to start services or clients on demand, and could be used for load-balancing or when booting a system.

3.2.8 The ServiceUI (user interface) project

Normally, a proxy object does not include any user interfaces (UI). The UI is implemented in the client software, and relies on the assumption that all proxy objects of a given type can use the same UI. This can be a limitation in dynamic environments, where different objects of the same type could benefit from having different UIs.

There is, however, a project going on in the Jini community that deals with coupling UI objects to proxy objects [26]. According to the specification, a UI must not necessarily be a graphical user interface, but could also be a speech interface, a text interface or any other user interface. This is shown in Figure 3.4, which is taken from the online specification [26].

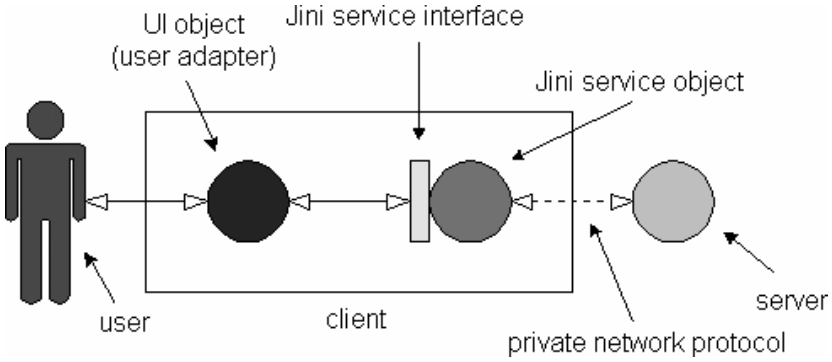

Figure 3.4 The ServiceUI specification

3.3 Tuple spaces and JavaSpaces technology

3.3.1 Introduction

In this section, we will take a closer look at the space-based communication paradigm and loosely coupled communication. We will start out with an introduction to tuple spaces before focusing on the JavaSpaces specification.

3.3.2 Coordinating concurrent activities with Tuple Spaces

«Tuple spaces have proven to be among the most fundamental and successful abstractions for coordinating concurrent activities». [27]

There are many ways in which one can make a parallel-processing system. The most apparent solution may be to coordinate and synchronize the processes on different processors through shared memory. This approach is often limited by both price, availability of hardware and lack of portability of software written for the specific hardware. Often, sharing the same physical memory is not an option for processes that are distributed across networked nodes.

When processors do not share the same physical memory, another apparent solution for coordination may be different ways of passing data between the processors. When this data exchange is done in an asynchronous manner, it is called messaging. In fact, many different standards and implementations of message-passing middleware exist for different programming languages and platforms. The Message Passing Interface (MPI) [28] is one standard for writing message-passing programs.

The Linda system, first published in 1985 by David Gelernter at the Yale University, is based on the Tuple Space shared memory principle. It takes a different approach for coordination of processes compared to message-passing. But as we shall se in chapter 8, a message-passing implementation may reside on top of tuple space shared memory middleware. Linda is a programming model and a set of libraries for simplifying the creation of parallel-processing systems. It is based on the tuple concept, which has its root in early research on artificial intelligence and blackboard communication architectures done by IBM. The term "blackboard communication" is used on communication that is based on a publicly readable and writable data repository.

Communication

Compared to message-passing (see Figure 3.5, taken from [29]), in space-based systems information is exchanged between processes in a non-direct manner. All information is copied between the processes and the Tuple space, and is not sent peer-to-peer. A Tuple space is a shared memory from which processes concurrently may read, write and consume data (see Figure 3.6, taken from [29]). Contrary to processes that communicate by passing messages, processes communicating through a Tuple space do not have to

exist at the same time to be able to communicate. The Tuple space functions as an intermediate data repository.

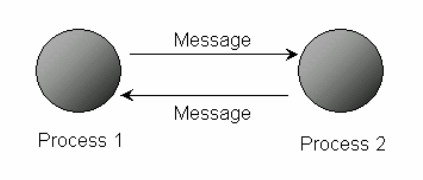

Figure 3.5 Basic message passing

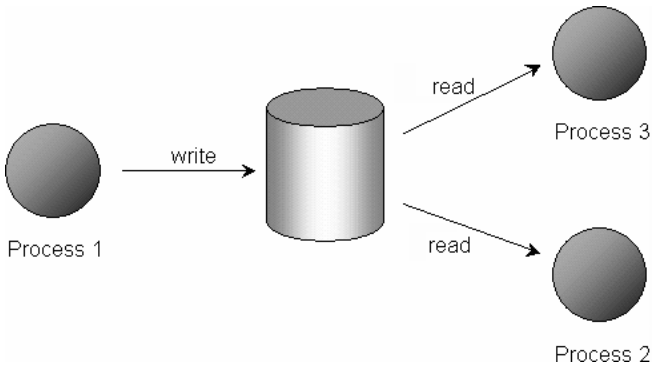

Figure 3.6 Space-based communication

Another property of tuple space-based communication is that processes do not need to know the exact location or address of the process they want to interact with. An *uncoupling in time, space and destination* can be achieved through *associatively addressed messaging*. By associatively addressed messaging we mean that processes use template matching to determine which tuples to read or consume from the tuple space. There is no need to know about any other physical addresses than that of the tuple space, but as we will see later in this chapter, using Jini eliminates the need to know any physical addresses. By uncoupling in time, space and destination [15] we mean the following:

Uncoupling in time:	Tuples exist independently from both sender and receiver. This enables processes to communicate even if they do not exist simultaneously.
Uncoupling in destination:	A sender does not need to know anything about the future use of a tuple, including the process that eventually receives the tuple.
Uncoupling in space:	Even though processes do not operate on the same address-space, it is still possible for them to interact in a machine-independent way. This is possible due to associative addressing.

The terms "uncoupled communication" and "loosely coupled communication" are used frequently throughout this thesis. These are just other ways to describe communication that is uncoupled in time, space and destination.

Tuples
A tuple is a vector of typed values or fields that are used for matching other tuples in the shared memory. For instance if a tuple A with the values <"ComputerA", 195, "ComputerB"> is written to the space, a read or take operation on the space with the tuple <String, int, "ComputerB"> as an argument, will return the tuple A (see Table 3.2). A formal field is a field that has a type, but no associated value. Various and more advanced types of tuple matching do also exist.

Table 3.2 Tuple matching

Regular Tuple	Formal Tuple	Match?
<"ComputerA", 195, ComputerB">	<String, int, "ComputerB">	Yes
<"ComputerA", 195, ComputerB">	<ComputerC, int, "ComputerB">	No
<"ComputerA", 195, ComputerB">	<String, int, String>	Yes

A Tuple space is often called a distributed shared memory because distributed processes are interacting with the space through various RPC mechanisms.

Tuple space and agent communication
The tuple space communication paradigm is often referred to as an ideal infrastructure for inter-agent communication. An agent framework based on tuple spaces is said to be robust, scalable and adaptive. It is said to be robust because one agent crashing will not bring the whole system down. Replication and mirroring of persistent spaces permits communication regardless of partial network and system failure. Scalability is achieved by adding new spaces and agents. Adaptability is simplified because agents may communicate without knowing each other's addresses. Agents may also communicate even if they are not running at the same time. Since the communication is asynchronous, anonymous and associative, a variable number of distributed agents can work together to solve a task.

3.3.3 JavaSpaces technology

Sun Microsystems' JavaSpaces specification [14] is based on the Linda programming model and adds Java objects to the tuple space communication model. In essence, the tuple space corresponds with the JavaSpace and the entities in the tuple space (the tuples) correspond with Java objects called entries. Interaction with the JavaSpace is done by RMI (see [9] and [10]). The matching of entries in a JavaSpace is much like the matching of tuples, as the matching is based on attributes in the Java objects, but the JavaSpaces programming model adds matching based on type as well.

When Java objects are written to a JavaSpace, their content is serialized together with an RMI-codebase annotation. Objects written to a JavaSpace stay serialized in the space

until a process does a read- or take-operation that matches the object. The read- and the take-operations both take a template as an argument. A template is an entry that describes the entries that the process is interested in. An exact match is not necessary, fields may be left empty when composing a template. When a match occurs and an object is read or taken from the space, its RMI codebase annotation is used to retrieve the actual implementation (class file) for the object. In this way, the JavaSpace can transfer objects with both state and behavior that may be deserialized by the client.

The JavaSpaces implementation from Sun Microsystems, Outrigger, is tightly coupled with the Jini technology. In fact the JavaSpaces implementation comes with Sun's Jini distribution as a Jini service. Because JavaSpaces is available as a Jini service it is possible to use Jini's lookup and discovery protocols to discover and interact with any JavaSpace available at any point in time. Operations on the JavaSpace are invoked through the operations on the Jini service proxy object, which implements the interface `net.jini.space.JavaSpace`. This provides a dynamic way of discovering and accessing available JavaSpaces.

3.3.4 The JavaSpaces interface

Figure 3.7 shows the methods in the `net.jini.space.JavaSpace` interface. Next, we will give a short introduction to these methods.

```
public interface JavaSpace {

    Lease write(Entry entry, Transaction txn, long lease)
             throws TransactionException, RemoteException;

    Entry read(Entry tmpl, Transaction txn, long timeout)
             throws UnusableEntryException, TransactionException,
                    InterruptedException, RemoteException;


    Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
             throws UnusableEntryException, TransactionException,
                    InterruptedException, RemoteException;


    Entry take(Entry tmpl, Transaction txn, long timeout)
             throws UnusableEntryException, TransactionException,
                    InterruptedException, RemoteException;


    Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
             throws UnusableEntryException, TransactionException,
                    InterruptedException, RemoteException;


    EventRegistration notify(Entry tmpl, Transaction txn,
             RemoteEventListener listener, long lease,
             MarshalledObject handback)
             throws TransactionException, RemoteException;

    Entry snapshot(Entry e) throws RemoteException;

}
```

Figure 3.7 JavaSpace interface

Write

The write operation writes an object of type `Entry` to the JavaSpace, where its serialized content is stored together with the RMI codebase annotation from the originating JVM. The operation also takes a transaction object as parameter. If the write operation is not a part of a transaction, the value `null` is used. The lease parameter is used to indicate how long it is preferable to keep the entry in the JavaSpace. The write operation returns a lease that must be renewed in order to keep the lease alive and prevent the entry from being garbage collected.

Read

The read operation returns an object of type `Entry` that matches the template argument. This is a blocking call that will not return until a match is found or a timeout occurs.

Take
The take operation is equal to the read operation with the only difference that it removes the entry from the space. If two processes try to take the same entry, only one will succeed. It will then be arbitrary which of the processes that succeeds.

Non-blocking versions of read and take
Non-blocking versions of the read- and take-operations also exist. They are called readIfExists and takeIfExists. It may seem strange to have a timeout parameter on the non-blocking versions, but the timeout parameter has a different purpose here. If the entry is part of a transaction, it will not be visible to other clients of the space until the transaction is completed. If a process tries to do a read- or take-operation on an entry that is part of a transaction, the method call will in fact block until the transaction is finished or the time specified with the timeout argument has elapsed.

Notify
The notify method is used in order to receive asynchronous notifications from the space when an entry that matches a specified template is written to the space. Jini's framework for distributed events is used, so the client of the space must register a remote event listener with the space. The method also takes a `MarshalledObject` as an argument. This object is returned upon notification in order for the client to simplify the process of determining which notification corresponds to which event registration.

Snapshot
The snapshot method is used for optimizing the interaction with the space if one or more entries are frequently used. By using this method it is possible to reduce the number of times the time-consuming serialization is performed.

3.3.5 Performance

Performance may be an issue when using JavaSpaces, since RMI and therefore serialization of objects is used for each message that is sent. The reader of the message will have to download class files that are not available in the local classpath to be able to deserialize the object. This process may be time-consuming, but the alternative ways of doing this, for instance parsing XML, are probably not much faster. This problem also depends on the size of the classes and the amount of classes that is needed, which may be reduced by intelligent programming and using the snapshot method of the JavaSpaces interface.

3.3.6 Summary

Loosely coupled communication through spaces may be even better suited for dynamic environments than the peer-to-peer proxy object interaction model. Using a space-based architecture can provide uncoupling in time, space and destination. This means that a client does not need to know where and with whom it is interacting. The service and the client may even be temporarily unavailable, but the message is still buffered in the space

until it is picked up. Of course, sometimes one would want use a tighter coupling when using a JavaSpace, for instance direct addressing. This is possible to achieve by using unique attributes on the entries written to the JavaSpace.

3.4 Jini and JavaSpaces security

3.4.1 Introduction

One of Jini's weak points has been the lack of a security architecture. There are several potential security problems in a Jini network, many of which could be solved by having a security architecture. With all the downloading of remote code that happens in Jini, it is essential to have the possibility to verify that downloaded proxy objects can be trusted. Both Jini and JavaSpaces are subject to many security threats, including denial-of-service attacks and spoofing attacks. Some of these risks can never be eliminated completely, some can be reduced by using a secure network infrastructure, and some might be removed by adding security features in the application.

3.4.2 Securing Jini software

There are two different areas where security mechanisms could be used in Jini software:

- Secure communication between proxy and service
- Secure download of a proxy object

How secure communication is realized between proxy and service is entirely up to the implementer of the service. He or she can use exactly the mechanisms wanted, whether it is cryptography or authentication. The only problem with this is that security libraries until recently have not been included in the Java 2 Standard Edition (J2SE) libraries. With the new 1.4 release of J2SE [39], these are bundled with the rest of the libraries, thus eliminating this problem for all computers that run this version. More details on the Java security model and the security libraries included in J2SE 1.4 are presented below.

If security is wanted when downloading a proxy object, a security architecture is most probably needed. The Davis project [40] targets this problem, and has released an early release of a Jini security model. More details are given below.

3.4.3 The Java Security model

Java has a security model [41] that allows different pieces of code to run with different permissions. Permissions are grouped together in a policy file, which again is used by a security manager. The security manager is the entity that actually performs the needed checks at runtime. An example of this would be the `RMISecurityManager` that must be installed before any remote code can be downloaded to a computer.

Mobile code is by default executed in a "sandbox", where it does not have access to any valuable resources, such as files. This means that when a proxy object is downloaded, by default it does not have access to any valuable resources, unless anything else is explicitly permitted.

The Java security architecture also makes it possible to use a signed object, which is an object that has a digital signature to prove its origin. This could be used to sign proxy objects, so that the client has a means of verifying that it has got a proxy object from a trusted service. Public Key Infrastructure (PKI) provides an infrastructure for using public and private keys (asymmetric cryptography) that are obtained and shared through a trusted authority.

The new release of the Java 2 Standard Edition (J2SE), release 1.4, includes several security features as part of its standard libraries. These are:

- JAAS (Java Authentication and Authorizing Service)
- JSSE (Java Secure Socket Extension)
- JCE (Java Cryptography Extension)
- Java GSS-API (Generic Security Services)
- Java Certification Path API

The three first of these libraries have existed as extension libraries, but are now bundled with the J2SE 1.4, thereby opening up for improved security with Jini. The Java GSS-API is an Application Programming Interface (API) for secure exchange of messages on top of a variety of underlying mechanisms. One such mechanism is Kerberos. The Certification Path API provides support for chained certificates.

3.4.4 The Davis project

Sun's Jini team is currently working on a security architecture in a Jini community project called Davis [40]. Export of secure proxy objects and placing restrictions on downloaded proxy objects are some of the problems it targets. The Davis project has released the Overture 0.04, which is an early release of the security architecture. A new RMI system, called Jini extensible remote invocation (JERI), is included. JERI provides more flexibility for customization than standard RMI. Support for using secure TCP (SSL/TLS), HTTPS and RMI/IIOP is also provided.

3.4.5 JavaSpaces security

Many of the issues that Jini faces also apply to secure use of JavaSpaces. Particularly, downloading of mobile code can be a threat. The technologies presented above can also be used for securing JavaSpaces, and the Davis project will make a secure version of Outrigger. Another solution could be that only "secure agents" are allowed to communicate directly with the space. These agents could be placed behind a firewall

with the agents functioning as proxies (in the Internet meaning of the word) forwarding only authorized messages to the space. This is illustrated in Figure 3.8.

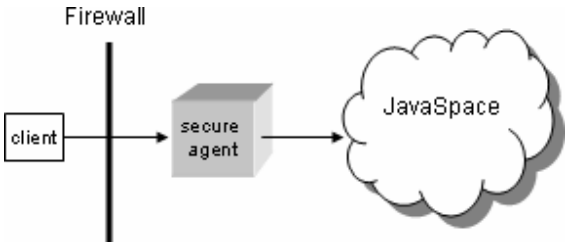

Figure 3.8 Using a secure agent with a space

The Yalta project ([42] and [43]) is a project that aims at building "A collaborative space for secure dynamic coalitions". It is based on JavaSpaces and the security mechanisms in J2SE 1.4. These two technologies are used to create a PKI solution for JavaSpaces. The project is sponsored by Defense Advanced Research Projects Agency (DARPA), which is the research and development organization for the US Department of Defense (DoD).

3.5 Summary

We have given a brief overview of Jini and JavaSpaces technologies. Both Jini and JavaSpaces are strong tools for creating distributed systems. Many Jini concepts are introduced, including leases, a distributed event model and distributed transaction model. JavaSpaces can provide loosely coupled object-based communication based on concurrent access to a distributed shared memory. By loosely coupled communication we mean uncoupling in time, space and destination. There are security challenges associated with the technologies covered in this chapter. Although the Davis project is not yet mature, it is a step in the right direction for securing Jini and JavaSpaces. Next, we will present three frameworks that utilize the Jini and JavaSpaces technologies.

4 Frameworks utilizing Jini and JavaSpaces

4.1 Introduction

The frameworks that we introduce in this chapter can be used to simplify development of systems that utilize Jini and JavaSpaces. In the next chapter, we will discuss how these frameworks could fit with JavaFrame-based technologies. Therefore, a brief introduction to the frameworks is given here. None of these frameworks are yet mature, but working prototypes exist of all the technologies. The three frameworks we have found during our study are:

- The Rio Architecture, [12] and [30]
- The Openwings Architecture, [13] and [31]
- The Spanish Inquisition or Scalable Infrastructure project, [22]

Rio and Openwings are somewhat similar, with much of the functionality in Rio being a subset of the Openwings functionality. Both architectures address a component model for deployment in service-oriented systems and can be used to simplify development of Jini-based applications. The Spanish Inquisition project, on the other hand, is an architecture that is based on JavaSpaces technology.

4.2 The Rio architecture

Rio is being developed by Sun, and it is basically a component model with Quality of Service (QoS) capabilities built on top of Jini. The components in Rio are called Jini Service Beans (JSBs). These beans can be loaded over the network and be instantiated in containers called Cybernodes. QoS attributes are used to describe the capabilities of Cybernodes and the requirements of JSBs. This provides an effective way of distributing components (JSBs) to the containers (Cybernodes) with the best matching capabilities. The Rio architecture also provides a management system with logging capabilities as well as a graphical view of the Jini context. Figure 4.1 (taken from [12]) shows a Cybernode that downloads JSB implementations based on JSB attributes. The Cybernode instantiates the JSBs and registers them with one or more Jini Lookup Services.

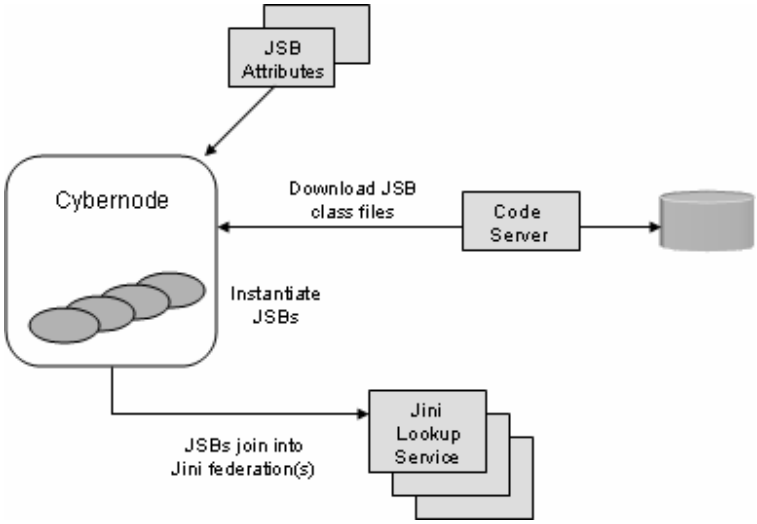

Figure 4.1 Rio functionality

4.3 The Openwings architecture

Openwings is being developed by General Dynamics Decision Systems (formerly Motorola IISG) and Sun, and targets both the military and the civil domain. The goal is to provide a Service-Oriented Programming framework for highly dynamic networked systems of software and hardware components [13].

The Openwings framework provides more features than Rio, including support for deployment, failover, clustering and security. These are all designed to ease the developer's task and thus let him or her focus on domain-specific issues.

Openwings also provides a container model, much like Rio's Cybernodes, but here Jini is only used as a plug-in in the Openwings container. Openwings aims to abstract over the actual service discovery mechanism that is used in a particular case, and there are plans to make plug-ins for JXTA [8], Bluetooth service discovery [32] and Web Services (Microsoft's .Net [33] and Sun's Open Net Environment (Sun ONE) [34]), among others. This means that an Openwings component actually may outlive Jini if a new service discovery technology proves to be better suited. It must, however, be mentioned that at the time of writing, Jini is the only supported service discovery plug-in.

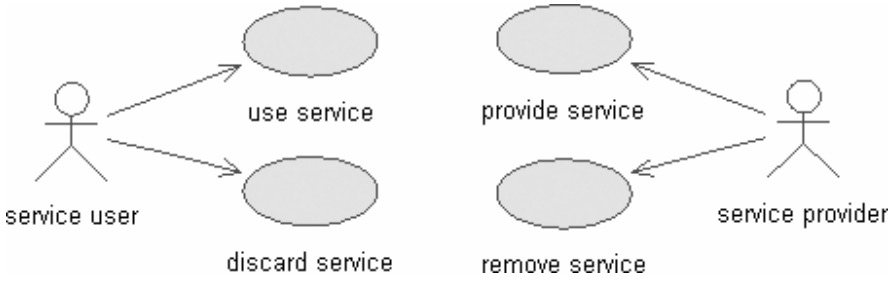

Figure 4.2 Openwings synchronous use case

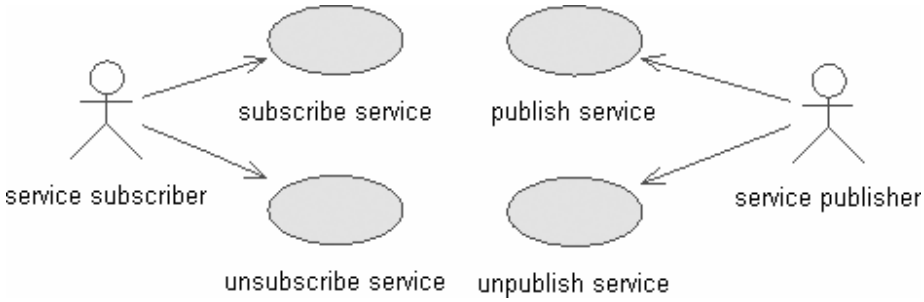

Figure 4.3 Openwings asynchronous use case

Figure 4.2 and Figure 4.3 show both the synchronous and the asynchronous use cases for Openwings. The figures (taken from [13]) show an abstraction over discovery mechanisms and over the lower level protocols used to access the service.

A notion called Architecture Description Language (ADL) is used to describe software architecture [35] in Openwings. ADL uses the notion of a Connector for communication between components. This is shown in Figure 4.4, which is taken from [13]. When Jini is used, this Connector would use a Jini proxy object. Both synchronous and asynchronous Connectors exist. Example technologies can be RMI or Java Messaging Service (JMS) [36], respectively.

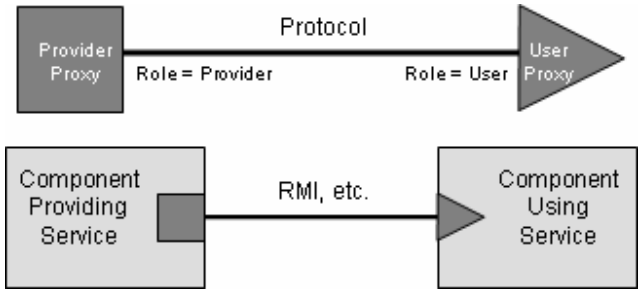

Figure 4.4 ADL connectors

4.4 Spanish Inquisition (SI), a scalable infrastructure

SI [22] is a space-based system that is being developed by Cisco Systems. According to [37], the system will «potentially allow an infinite number of devices/users to attach to a communication architecture and fulfill its mission under severe time restrictions in an almost real-time environment». SI builds on Jini and JavaSpaces, and is based on agents. These agents can monitor the JavaSpaces and make decisions, such as whether to forward messages to other JavaSpaces. SI's goal is to provide a highly available and scalable infrastructure by ordering JavaSpaces into a single "virtual space". Cisco claims to have performance measures that confirm linear scalability.

All communication in SI goes through one or more spaces, which are persistent and may be replicated. This can provide a backup of messages if a space, client or service temporarily goes down.

In Figure 4.5, the most important components in the SI architecture can be seen. Community Services keep track of spaces in a Jini Community, which is the same as a federation. A Community Service also maintains a list of all the agents in the community.

A service called a "betweener" forwards messages to spaces in other communities. The replication agent can be used to ensure that messages are not lost when a space goes down. This is done by writing messages to more than one space, preferably on another computer. The morph agent is a generic agent which can do different types of work based on the events it receives. The actual work is implemented in an object which is downloaded by the morph agent. Finally, there are two more types of agents in Figure 4.5, the double agent and the universal double agent. These deal with computers or devices outside the SI system by translating their protocols into the protocol that is used to communicate with the space.

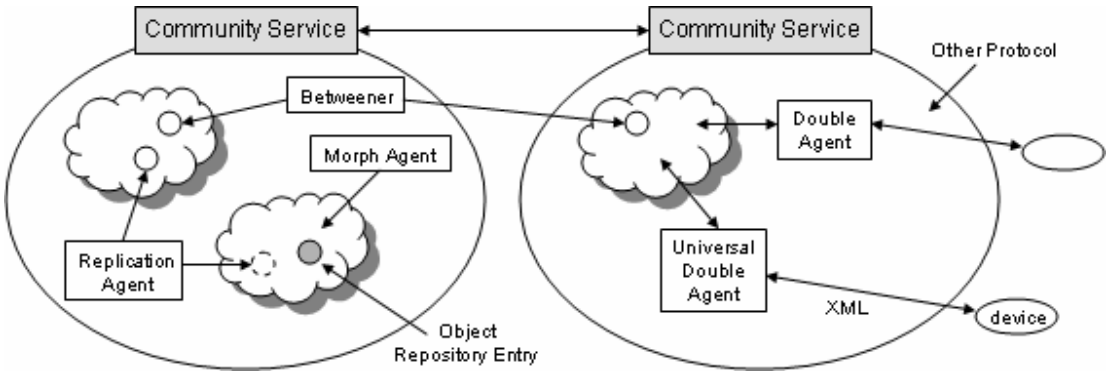

Figure 4.5 Partial figure of the SI architecture

4.5 Summary

This brief overview of Rio, Openwings and SI has shown that interesting work is being done in the Jini and JavaSpaces field of distributed computing. All three frameworks can be used to make it easier for developers to make effective use of Jini and JavaSpaces features. While Rio builds on Jini, Openwings uses Jini as a service discovery plug-in in its service-oriented architecture. SI, on the other hand, builds mostly on the JavaSpaces technology.

In the following chapter we will see that these technologies might be useful together with JavaFrame-based technologies when it comes to distribution.

5 Using Openwings, Rio and SI as middleware for JavaFrame-based systems

5.1 Introduction

The thesis definition suggests that an evaluation of Openwings, Rio and SI as middleware for Ericsson NorARC's technologies for service creation will be performed. Each of these technologies may prove to be quite powerful in dynamic environments, especially in terms of deployment functionality and reliability. By integrating these frameworks with the JavaFrame-based technologies, deployment and distribution problems might be possible to handle in a simplified way. In this chapter, we will evaluate if and how integration could be done.

From what we understand from the quite limited documentation of Rio, its functionality is merely a subset of the Openwings functionality, even though Rio includes Quality of Service (QoS) aware deployment. We will therefore partly treat these two architectures together when discussing their use as middleware together with Ericsson's frameworks. While Openwings' goal is to provide a general architecture for service-oriented technologies, Rio focuses solely on Jini.

5.2 Using Rio and Openwings with Ericsson's service creation technologies

Both Openwings and Rio are based on the concept of a container that provides system services. The most important of these services is service discovery. The thought behind Rio is to hide complex Jini functionality as container services that can be used by hosted components. Openwings' mission is to provide an abstraction layer for many different service discovery technologies, not only Jini. Openwings and Rio components are supposed to take advantage of the provided middleware, enabling them to adapt to change in an easier way. Other Openwings services include component lifecycle management, component deployment, fail-over, live-updates, mobile components/agents and clustering. Although Rio introduces QoS when it comes to deployment of services, the deployment model is essentially the same for both architectures since Openwings container services are meant to provide the same QoS functionality. Openwings is, however, richer in documentation and implementation.

There are several reasons why technologies like these should be considered used together with Ericsson NorARC's service creation technologies. Even if they are not used, they might have concepts that are worth taking a closer look at.

The main reason for considering the use of Rio or Openwings as middleware is probably to enable JavaFrame-based components to dynamically, during runtime, find and use each other's services. It would also be possible to use the Openwings architecture to

enable JavaFrame-based components to discover and interact with non-JavaFrame services, like Web Services. Other reasons why one possibly would try to fit a JavaFrame-based component inside an Openwings component would probably be to benefit from advanced deployment, clustering and fail-over functionality.

Our impression is that Ericsson NorARC until now mainly have focused on advanced software patterns and architectures for service modeling and service logic, and not that much on deployment issues. Although deployment functionality based on deployment descriptors now is included in ActorFrame, implementing advanced deployment functionality like the ones found in Openwings or J2EE application servers is a complex task. Openwings includes a "control shell" and both Rio and Openwings include graphical "explorers" that can be used to remotely deploy packaged components on different nodes over the network. By using Rio or Openwings with JavaFrame-based technologies, one could benefit from this advanced deployment functionality. It would also be possible to take advantage of Rio for QoS-aware deployment for JavaFrame-based components, e.g. for deploying components on the servers with the lowest load. As described in section 3.2.7, instantiation of new services is supported in the standard Jini libraries, but this may be simplified by using a container for life-cycle management.

5.3 Openwings-specific issues

5.3.1 Threads

Compared to J2EE environments hosting EJBs, the Openwings component model seems to be better suited to host JavaFrame-based components. Unlike the EJB component model, the Openwings component model does not place any restrictions on the component threading model. For instance, an EJB component is not allowed to host/spawn its own threads. As mentioned in chapter 2.2.1, each JavaFrame application has one or more Schedulers, each associated with a thread. Schedulers manage the message queues on the state machines. This is important to have in mind if a JavaFrame-based component with its own thread management is supposed to fit inside another component model, possibly with a conflicting threading model. However, this does not seem to be a problem with Openwings. Actually, one of the original motivations for creating the Openwings architecture was that the EJB component threading model was too restrictive [38], and was not seen as a good fit for the Openwings problem domains (advanced telecommunication and military systems). We have not been able to find out from the documentation how Rio's threading model fits with JavaFrame.

5.3.2 Openwings connectors vs. JavaFrame mediators

Openwings' notation of asynchronous services and connectors is in the spirit of JavaFrame's asynchronous communicating state machines and mediators. In fact, JavaFrame's mediators and Openwings' connectors appear to be two solutions for the same problem. Both mediators and connectors are interaction interfaces that are meant

to improve independence between different parts of the system. This allows for easier reuse of system components and development activities running in parallel.

Openwings' connector services are said to enable Protocol Independent Programming (PIP) and Protocol Independent Design (PID). PIP and PID focuses on implementation and design independent of lower-level RPC-mechanisms and transport protocols. This is much like the thought behind JavaFrame's ProtocolMediators, which hide low-level communication protocols. However, it is important to realize that neither PIP nor PID try to hide application-level service protocols. For example, using PIP or PID does not hide a protocol for message exchange between two JavaFrame state machines or the semantics associated with a service interface.

So what is the difference between JavaFrame's mediator and Openwings' connector? Mediators are interaction interfaces used both on the edges of the system and between different internal system entities. An example of an edge-mediator could be an RMI-mediator, but mediators are also be embedded deeply in the system logic within one JVM, enabling composition of active objects inside a system component.

Openwings' connectors are used solely on the edges of a component, representing a logic entity capable of either producing or consuming a service. Connectors are used for connecting components that provide services with components that use services. They are closely related to Openwings' component services, addressing component connectivity. By using the component services, a component is able to discover and use another component's services without knowing which type of connector it has to use (IIOP, JMS, RMI, etc.). This forms the basics behind Service-oriented programming (SOP) and Service-oriented architecture (SOA). The use of Jini and mobile code are key enablers for SOP in Openwings. When using JavaFrame-based applications today, one has to compile with a specific set of mediators (RMI, CORBA etc.), not being able to change the mediator's low-level communication protocol at runtime.

5.3.3 Alignment of Openwings and JavaFrame

What would it then take to align JavaFrame-based technologies with the Openwings architecture? It basically comes down to grouping JavaFrame-based system components into single logical Openwings components that produce and consume services at the edges of the components. There could possibly be a one-to-one relation between a mediator and an asynchronous Openwings connector. To align these two concepts, one would have to adapt a view of the mediators on the edges of the components as service interfaces that either produce or consume a service. This service interface would have to be an asynchronous message interface capable of transporting JavaFrame messages.

There would have to be a standard for describing the asynchronous service interfaces mentioned above, enabling the components to find a matching service-mediator at runtime. The task of describing these asynchronous service interfaces is probably a

greater task than one might first think of. An asynchronous message interface described with a Java interface is ambiguous in this case. It does not say any thing about the correct application level protocol associated with the service interface. Service attributes could be used to describe the asynchronous service, but one would still need a standard for these attributes. A component using another component's service without knowing the correct application level protocol and semantics would most probably lead to failure. Description of the semantics of an interface is probably one of the great challenges when it comes to components dynamically forming into systems. A discussion of these issues is presented in chapter 7.9.

5.3.4 ActorFrame and ServiceFrame

How does this then fit with ActorFrame and ServiceFrame? Both ActorFrame and ServiceFrame are designed to be more lightweight and faster than what would probably be possible to achieve with for instance EJB. JavaFrame's footprint is actually as low as about 60 KB, enabling it to be used on everything from the largest servers to some of the smallest embedded Java processors.

Components within ActorFrame and ServiceFrame (e.g. Actors) rely on extensive inter-component messaging, partly due to ActorFrame's Role-request protocol and the fact that all communication is based on asynchronous messaging. JavaFrame's efficient design, low thread usage and its ability to run several components (e.g. Actors) within one JVM means that the framework is efficient.

Distribution of ServiceFrame components that rely on heavy inter-component messaging across different JVMs should only be done when absolutely necessary. This distribution introduces a bottleneck and substantial latency compared to JavaFrame messaging within one JVM. Taking this into account, splitting up the components within ActorFrame and/or ServiceFrame and wrapping them as Openwings components would probably not be very lightweight and no elegant solution. It would probably be better to organize larger pieces of service logic into Openwings components that produce and consume services. This would enable them to form into systems at runtime. Here, "larger pieces of service logic" could possibly be a whole instance of ServiceFrame delivering services at its edges or a collection of ActorFrame actors working together to provide services.

Another important consideration is how system components are designed and implemented with regards to change, e.g. how they react when services they use become unavailable. The Openwings component model does not aim to hide the effects of change and system failure. Rather, it tries to expose these changes in a simplified way for the developer to handle. This enables the developer to create components that react to change, for instance by searching for a new service that matches the service that went down. The developer himself/herself has to decide how to recover from a lost session. The Openwings architecture has a non-static view on the world, forcing the developer to

decide how to handle change. If the Openwings architecture is to be used for distribution of JavaFrame-based system components, this has to be taken into consideration.

5.3.5 Hiding failure behavior or not?

How components handle change is probably something that should not be handled and hidden solely at mediator-level. The effects of change should be exposed by the mediators and handled by a set of JavaFrame state machines at the "application-level". This implies that for ActorFrame and ServiceFrame system components to benefit from the Openwings architecture, some extra functionality will have to be added. Since much of the service logic in ActorFrame and ServiceFrame is located within one JVM, it has not been necessary to make the system components capable of handling the case that other system components become unavailable. With Openwings, this has to be taken into consideration when designing components from the ground up. Enabling JavaFrame-based components like ActorFrame actors to react to change is most probably possible and may probably be handled in an elegant way by using composite states.

5.3.6 Possible limitations of Openwings

While Openwings is an architecture that focuses on how components find each other's services and adapt to change, it says nothing about how the service logic of these components should be modeled and implemented. This is exactly what the JavaFrame-based technologies focus on. Therefore, Openwings and JavaFrame-based technologies could possibly be a powerful combination. JavaFrame's mediator concept could be extended and possibly become more powerful by investigating Openwings' more "service-oriented" and adaptable form of component interconnection.

Even though the use of a container provides several interesting features, the actual connections with services are still of the peer-to-peer type. This means that if a service fails, a new service discovery or lookup has to be done. When using a space-based architecture, even this possible weakness may be reduced or removed.

5.4 Using SI with Ericsson's service creation technologies

As indicated above, the container model provides many interesting features. But the interaction model is still of the peer-to-peer type. By interaction model we mean the way in which the communicating parts interact. With Openwings and Rio, the model is very similar to the Jini model where a client finds a service and contacts it directly. SI however, uses a completely different interaction model. It is a space-based system where agents represent both services and clients. The interaction goes through a space, which is an intermediate storage for messages. A space may also be replicated and persistent and thus can be fail-tolerant.

As we have seen in chapter 4.4, SI seems to offer a powerful distributed communication platform. The reason for adapting such a space-based communication model would be to benefit from the loosely coupled asynchronous communication explained in chapter 3.3. How then, could JavaFrame-based system components benefit from using SI?

Since the SI architecture with all its agents is already defined, using SI together with Ericsson NorARC's frameworks may be difficult. A mapping is needed between the components of the ServiceFrame framework and SI. This could be a problem since SI in this setting should only provide middleware functionality, not force ServiceFrame concepts to be altered unnecessarily.

However, used as a platform for communication SI may prove to be a powerful tool. In its current state, SI is poorly documented and the implementation is not complete. If deciding to use SI as a communication platform for JavaFrame-based systems, an agent-based approach would be needed. The work of creating an agent-based architecture is partly done in our effort in building an independent architecture, which is presented in chapter 8.

We decided to build a space-based system of our own, building upon the properties that we have seen that space-based systems can offer. Inspired both by ActorFrame, ServiceFrame and SI, a conceptual space-based architecture has been developed. The architecture and a prototype are described in chapter 8 and 9.

5.5 Space-based model or container model?

So far we have discussed whether the container model or a space-based model should be used. But why not combine the two approaches?

Combining a container model with a space-based architecture may actually be a quite good idea. By having agents belonging to one or more spaces running in containers, the agents would be able to benefit from the container services. This might lead to system independence for agents, which could be compared to components when using such a model. From an Openwings point of view, a space-based communication model may be offered by wrapping a JavaSpaces service as an Openwings component. This Openwings component would then provide a distributed coordination service.

The combination of the container model and the space-based model would still need an architecture like the one we present in chapter 8. Whether our prototype is going to be used or not, the architecture still provides a starting point for adapting JavaFrame based technologies to a space-based approach.

5.6 Summary of framework evaluation

In this chapter, we have focused mostly on Openwings and SI because Rio turned out to provide only a subset of the functionality in Openwings. For JavaFrame-based technologies to benefit from the Openwings architecture or way of thinking, the greatest challenge is that system components have to be designed explicitly for handling change at runtime. Handling change is probably something that can not be handled solely at "mediator-level" (e.g. hidden by a mediator from the higher level application logic) but will influence "application-level" system design and implementation. This is also the case with Jini, as we shall see in the next chapter.

Benefiting from SI probably implies that an agent-based approach for system design is needed. An agent-based approach will most probably not conflict with ActorFrame/ServiceFrame and certainly not with JavaFrame. In chapter 8 we will present our own ideas for an agent-based solution.

This evaluation has shown that there are interesting similarities between JavaFrame-based systems and Jini- and JavaSpaces-based systems. Valuable functionality can be achieved from integrating these frameworks with JavaFrame-based systems. In the following chapters, we will explore how such integration may be done from the ground up.

6 Introduction to conceptual solutions

This chapter is an introduction to several conceptual solutions that we have developed, focusing on integration with JavaFrame-based systems. In chapter 7, we present conceptual solutions that are based on Jini and JavaFrame concepts. Our space-based architecture is presented in chapter 8, and combines JavaSpaces and JavaFrame concepts.

This project has focused on how Jini and JavaSpaces might be used for distribution of systems based on Ericsson NorARC's technologies for service creation. These technologies are at the moment JavaFrame, ActorFrame and ServiceFrame, as explained in chapter 2. When creating ServiceFrame, the creators probably had a static and safe server environment (service network) in mind. In such an environment services are deployed on networked nodes with very low downtime. For instance, services have been hosted within a service execution environment and accessed through SMS, HTTP or WAP.

Special JavaFrame mediators for communication via Bluetooth, RMI and CORBA have already been made and mediators for SOAP/Web Services are planned. It is possible to use for instance RMI mediators to distribute a JavaFrame system across multiple networked nodes running separate Java Virtual Machines (JVMs). Services are envisioned to be accessed from a multitude of different terminals and devices with varying capabilities via several different bearer technologies.

Services in ServiceFrame are composed of different actors that play different roles. Interaction is done by passing messages. In current versions of ServiceFrame all actors are located within one execution environment and are communicating inside the same JVM.

It is possible to statically set up RMI connections with RMI mediators between actors in different JVMs, and RMI mediators have proven successful in a number of situations. However, as services are supposed to be delivered in a more heterogeneous and ever-changing environment where both services and clients come and go across different platforms and bearer-technologies, more dynamic and adaptable interconnections are needed. In such an environment, basic RMI mediators and similar mediators will probably be too static to cope with the rapid changes. Components must probably also be designed explicitly to cope with change from the ground up.

Our vision of the future is that services may not only be hosted by powerful application-servers in a safe server environment, but deployed across everything from cheap system-on-chip processors embedded everywhere to the server environments we have today. As we already have seen in section 3.2 and 3.3, both Jini and JavaSpaces offer several interesting facilities to help enable this vision. Jini enables applications to discover available services, access them through mobile code and adapt to change.

JavaSpaces as a Jini service combines these properties with an object-oriented distributed coordination framework.

For JavaFrame-based applications to be able to utilize Jini and JavaSpaces, some kind of specialized mediators are needed. In the following chapter, we will present several conceptual solutions we have developed to show how Jini could be used together with Ericsson NorARC's technologies.

7 Conceptual solutions based on Jini

7.1 Introduction

In chapter 5 we looked at architectures and frameworks built on top of Jini, thereby providing an abstraction layer. In this chapter, we will take a closer look at the details of how JavaFrame-based technologies could benefit from using Jini. As seen in chapter 3.2, Jini has several beneficial features when it comes to developing advanced distributed systems. We have therefore developed four conceptual solutions that utilize the Jini architecture to distribute JavaFrame-based technologies. In the first solution, Jini is utilized to enable JavaFrame mediators to dynamically discover and download the correct protocol for interaction with a corresponding set of mediators. The second solution utilizes Jini to publish JavaFrame-based services to non-JavaFrame clients. In the third solution, JavaFrame software is embedded in Jini proxy objects. Finally, we present a solution where a role provider proxy can be used to dynamically download roles.

7.2 Using Jini between JavaFrame environments

JavaFrame mediators are used for communication between active objects. For bidirectional communication, two mediators are needed, one in each direction. In principle, mediators can communicate using any given protocol, as long as there is an implementation available. Jini itself is not such a communication protocol, although the Jini specification includes a set of protocols. Rather, Jini is a technology that can be used by mediators to obtain any protocol [44] that is needed to communicate with another mediator. By using Jini, a client based on JavaFrame or any other JavaFrame-based framework would not need any compile-time knowledge of the protocol used by a mediator, as long as it knows what kind of service or service mediators it wants to use. The actual proxy object, with its specific communication protocol, could be requested by the Jini-enabled mediator at runtime through the service discovery procedure. This is shown in Figure 7.1.

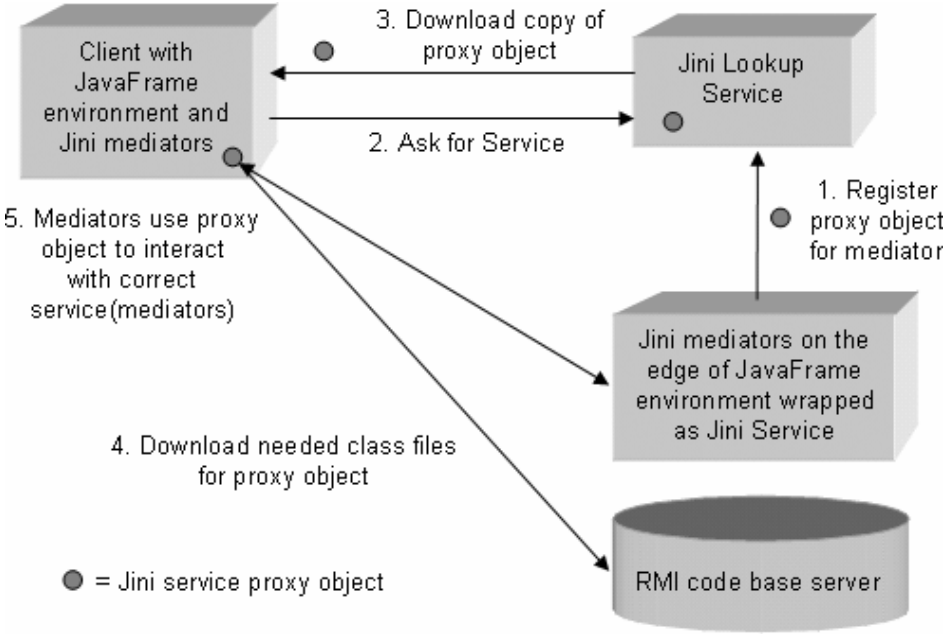

Figure 7.1 Mediators capable of interacting through a Jini service proxy object

To make clients able to do download its proxy object, a JavaFrame-based service must behave as a Jini service. This is often referred to as wrapping something as a Jini service. What this means, is that the JavaFrame component must be able to locate at least one Jini Lookup Service, where it must register a proxy object for the service. When the proxy object is registered at the lookup service, a client can look it up, that is, check if someone somewhere offers a specific service. This can be specified by both Java interface and attributes. If there is a match, the client's mediators download the proxy object from the lookup server. Now the client's mediators can use the downloaded proxy object to communicate with the other JavaFrame peer's set of mediators, for instance mediators belonging to a ServiceFrame service.

A Jini service proxy object can provide asynchronous bidirectional communication based on Jini remote events. In order for a JavaFrame-based client to have bidirectional communication with the service, a pair of "service-mediators", one in each direction, might be a good idea. This pair of Jini-enabled "service-mediators" would then share one Jini service proxy object. The client does not need to have any implementation, nor any prior knowledge about the low-level communication protocol that is used. This choice is made entirely by the service itself, and may result in a more predictable situation for the service. Messages are forwarded, using the correct protocol embedded in the proxy object. In principle, the protocol used could be anything from a proprietary protocol to HTTP, CORBA or RMI. However, since JavaFrame messages are supposed to be sent transparently from a JVM to another, RMI might be the best choice.

Another interesting possibility one can get by using proxy objects is that the service provider may embed a JavaFrame environment in the proxy object. Based on the capabilities of the client, system logic could transparently be distributed to the client. This could be done to different degrees, making clients "thicker" or "thinner". As an example, if the client has free system resources, parts of the service could actually execute on the client side without the client knowing anything about this distribution. The client would continue to use the same set of mediators in the same way as before. As we shall see in section 7.4, embedding a JavaFrame environment in the proxy object may also be done to enable non-JavaFrame clients to act as JavaFrame peers. This could be done transparently to the client.

When comparing these conceptual Jini mediators with already existing types of mediators, two things become clear. The mediators that provide a service have to handle the process of finding a lookup service and registering a proxy object, while the mediators that use a service have to go through the service discovery process. If the connection is lost, the Jini mediators will have to look for an equivalent service and possibly also restore session state. This functionality is not provided by Jini itself. Much like what we saw with Openwings in chapter 5.3, this functionality has to be handled by the system logic that uses the mediators. Most likely, a special set of JavaFrame messages have to be developed to handle Jini-specific functionality in the mediators. It might also be the case that mediators that provide a service are different from mediators that use a service.

When using a set of Jini mediators the interface on the Jini service proxy object should provide a bidirectional asynchronous message interface. Often, the interface itself can not fully describe the semantics for interaction with the service and corresponding set of mediators. As we shall se in section 7.9, these interfaces have to be described in a way that ensures that system components do not start to interact with each other without knowing each other's correct semantics. This could be done by using attributes in an expressive way.

7.3 Jini for service distribution to non-JavaFrame clients

Let us consider the case when the client is non-JavaFrame and the service proxy object does not include any JavaFrame functionality. This might be the case when ServiceFrame is supposed to deliver services to clients that are capable of interacting with Jini-based services but do not contain a JavaFrame "runtime environment". The problem with this approach is that JavaFrame messages can not be delivered asynchronously and transparently end-to-end. Somewhere in the actual service the method calls by the client on the service proxy object have to be mapped to JavaFrame messages. This is not a very elegant solution compared to our next idea which uses JavaFrame messages end-to-end even though the client does not already host a JavaFrame environment.

7.4 JavaFrame embedded in Jini proxy objects

In a dynamic environment where clients come and go, it is reasonable to assume that not every client is JavaFrame-enabled. Jini could be used to give these clients the opportunity to connect to a JavaFrame-based service (e.g. a ServiceFrame service) by transparently downloading the needed JavaFrame environment, embedded in the proxy object. Classes forming this JavaFrame environment would have to be available for download on an RMI codebase server.

A service user interface could be provided together with the proxy object as described in chapter 3.2.8. Coupling a proxy object and an UI object opens up many interesting possibilities. Services could have different user interfaces based on location, or a role could have an associated UI based on the type of terminal it runs on. The JavaFrame service logic inside the proxy object could provide functionality associated with the user interface.

Communication between client and service could be done for instance by using RMI. The mediators on the edge of the JavaFrame environment downloaded by the client interact with a corresponding set of mediators at the edge of the service provider. This solution, shown in Figure 7.2, could provide a unified platform, even in dynamic environments with new clients arriving all the time.

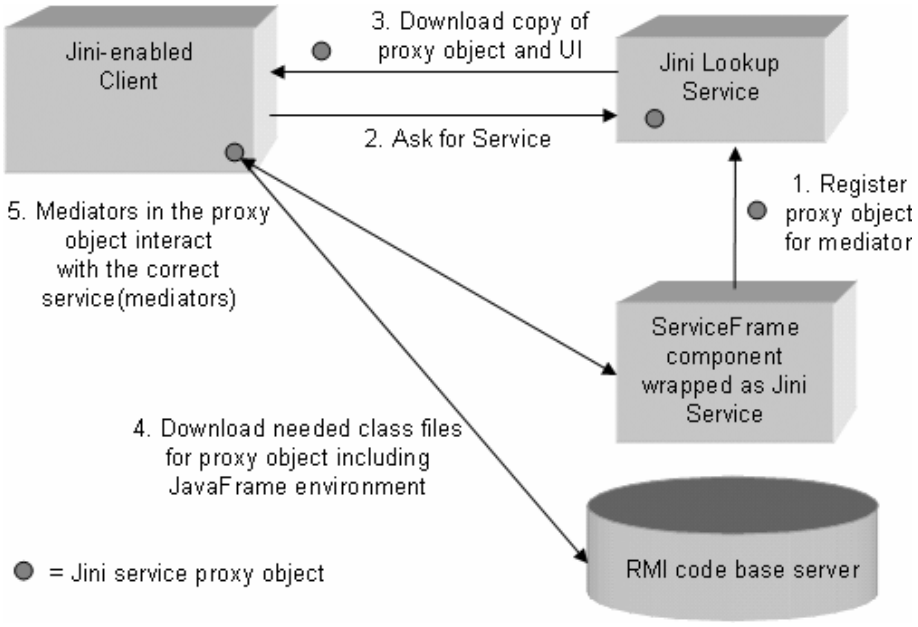

Figure 7.2 Download of JavaFrame environment and UI

7.5 Obtaining roles dynamically

A flexible solution could be to create a service that allows clients to download their role dynamically. Clients could then download a "role provider proxy", which would include the mediators and the protocol needed to obtain a role, as shown in Figure 7.3. This would need a standardized environment like ActorFrame for playing such a role and standardized semantics for querying the "role provider service". Some sort of verification that the role actually is possible to play would also be needed.

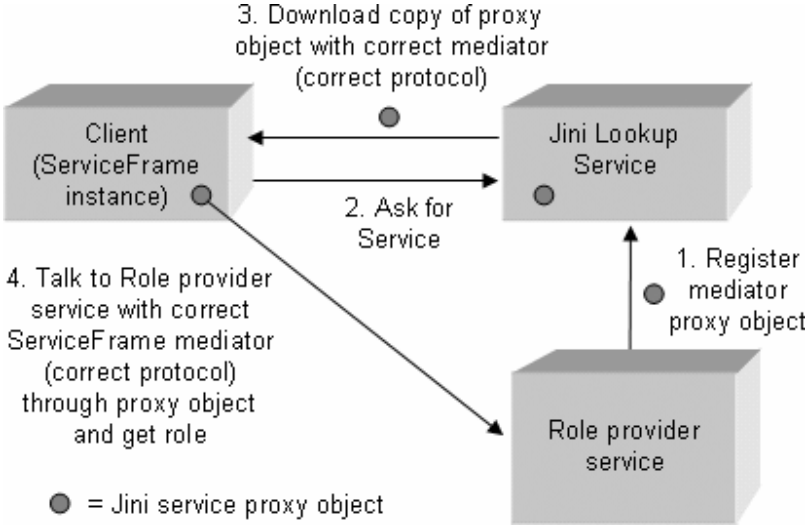

Figure 7.3 Using a role provider service

7.6 Updating software

When it comes to updating protocols or mediators, using Jini can also provide flexibility. Instead of having to update software in all clients, as will have to be done in most other systems, the service can simply write a new proxy object to the lookup service. Polymorphic principles would allow developers to replace or extend the implementations that are exported to the Jini Lookup Service, and the client will still be able to do lookup based on interface and attributes. Any new clients will now download the new proxy object, with its new behavior. The existing connections that are still using the old proxy object will eventually be closed as their leases expire. To be able to continue to use the service, clients using the old proxy object will have to do a new lookup, thereby downloading the new proxy object. The software update will by now be complete. Doing updates this way will need support for two different proxy objects by the service during a transfer period.

7.7 Failure model – Leases

The ideas presented above are all very interesting, but what if there is a failure somewhere in the system? As explained in chapter 3.2 about Jini, the whole Jini architecture is based on leases. If a lease expires or is cancelled on the Lookup server, the proxy object can not be downloaded by new clients. Any new clients entering the environment therefore can not use the service. They will have to lookup an equivalent service. This is often referred to as a self-healing network.

Clients that already have a session with the service, however, will not be able to continue. An exception will be thrown in the proxy object, indicating that something is wrong with the communication. There are two ways that this problem could be handled. One is to implement a policy in the proxy object for what to do when an error occurs. This implies that it actually is the implementer of the service that decides what to do, which is often not satisfactory. How does he or she know what the client wants to do at runtime?

The other solution to this problem is to let the state machine that uses the mediator implement its own policy for what should be done when a problem occurs. It could for instance ask a user if it should do a new lookup to find another "healthy" service that has similar capabilities as the failed service. However, the most elegant solution may be if the process of finding a similar service is transparent to the user.

In the cases where using proxy objects implies peer-to-peer communication, a failure can be a difficult problem, especially if reliability is to be guaranteed. It is a problem with established peer-to-peer connections generally that if one peer fails, the other peer must look for a new peer that can provide the same or similar functionality. As is explained in chapter 3.3 about JavaSpaces technology, it is possible to lessen the dependency on single peers and increase flexibility by communicating through a space.

7.8 Roaming between IP networks

Another scenario where a JavaFrame-based client could benefit from using Jini is when a client is roaming between two IP networks, and an existing connection with a service is lost. An example of roaming between two IP networks could be a client moving between two Wireless LANs. Functionality could be implemented on top of Jini that would listen for exceptions that indicate a lost service. When this happens, the client can do a new lookup, trying to obtain a new proxy object from an equivalent service. The new proxy object can then be used to communicate with the newly found service, perhaps also continuing its previous session. Continuing a session means that session state would have to be re-established. This would require specialized logic in both client and service software.

It is interesting to note that the new proxy object actually could have a completely different implementation than the first service that was used. This property could be used to help the client adapt to any specific constraints placed on the software by the new network.

Although this use of Jini could provide much flexibility, it depends on the service how session state could be re-established. In many cases, the client will have to establish a new session and redo initialization procedures to recover its state.

7.9 Lookup with asynchronous interfaces

An interesting problem that becomes clearer when using asynchronous messaging interfaces is that interfaces for different services can all have the same methods, even if they do not have the same behavior and semantics. For instance, the "send" and "receive" methods of two different services can expect two completely different messages as input, containing different payloads with completely different application level protocols. An interface itself says little about the kind of system that hides behind its interface and the semantics for correct operation that is associated with this interface. This means that it could be meaningless to do lookup based on Java interface only. As already stated above, components forming into systems without knowing each other's correct semantics will most probably lead to failure.

One way to solve the problem described above could be to name the interfaces (or tag them, extending another empty interface), after the service it provides. Clients using the interface must then have prior knowledge about different interfaces and their semantics. Based on the interface name of the proxy object and on prior agreement upon a set of rules, the client uses the proxy object in the correct way. A better way to solve the problem is to use attributes in a more expressive way, describing the correct semantics for the interface. This would also need some kind of prior agreement and common understanding of the attributes, but this is outside the scope of this thesis.

7.10 Summary

We have seen that Jini can be an effective means for distributing JavaFrame-based services. Many concepts that provide flexibility are described, including protocol independence and downloading of necessary software. We have introduced the new concept of a Jini-enabled mediator that allows JavaFrame-based components to cope with change. In some cases, however, the peer-to-peer interaction model can prove to be a limitation. This limitation is dealt with in the following chapter, where all communication goes through a JavaSpace.

8 A conceptual space-based architecture

8.1 Introduction

This chapter, along with chapter 9 and 10, presents our main contribution. To fully exploit Jini and JavaSpaces, we have designed our own conceptual space-based architecture. In the previous chapter, we described how Jini could be used for distribution of JavaFrame-based services. This chapter presents a conceptual space-based architecture where JavaSpaces technology is used on top of Jini. Several ideas are presented to show the benefits of using such an architecture. In chapter 9, we present our SpaceFrame prototype, where we have implemented a subset of the concepts described in this chapter. In chapter 10, we present an example application to show how our conceptual architecture works, and how the SpaceFrame prototype can be used.

8.2 Goals

JavaSpaces is a distributed coordination framework based on an "object flow" in and out of a distributed shared memory. Our idea was to find out if this "object flow" might consist of JavaFrame messages and objects containing role behavior. We also wanted to see if the principles of uncoupling in time, space and destination could be used successfully together with JavaFrame-based systems.

To ensure that one or more JavaSpaces could be found, providing an "always-on" functionality, we wanted to use JavaSpaces as a Jini service. By doing this, it would be possible to transparently distribute JavaFrame-based components across different networked nodes, regardless of changing IP addresses and services coming and going all the time.

In current versions of ServiceFrame, the class files that contain the actual implementation of a role have to be located in the classpath of the executing JVM. We wanted to use Jini and JavaSpaces to transfer objects with behavior (mobile code) between different JVMs in order to distribute role behavior. In other words, we wanted to use JavaSpaces as middleware for inter-actor communication.

8.3 Architecture description

8.3.1 High-level description

Our proposed architecture is heavily based on Jini and JavaSpaces and also inspired by the SI project described in chapter 4.4. All communication is based on asynchronous associatively addressed messaging through a JavaSpace. An illustration of the conceptual architecture is shown in Figure 8.1. We will give a brief explanation here for overview, and go into more details later.

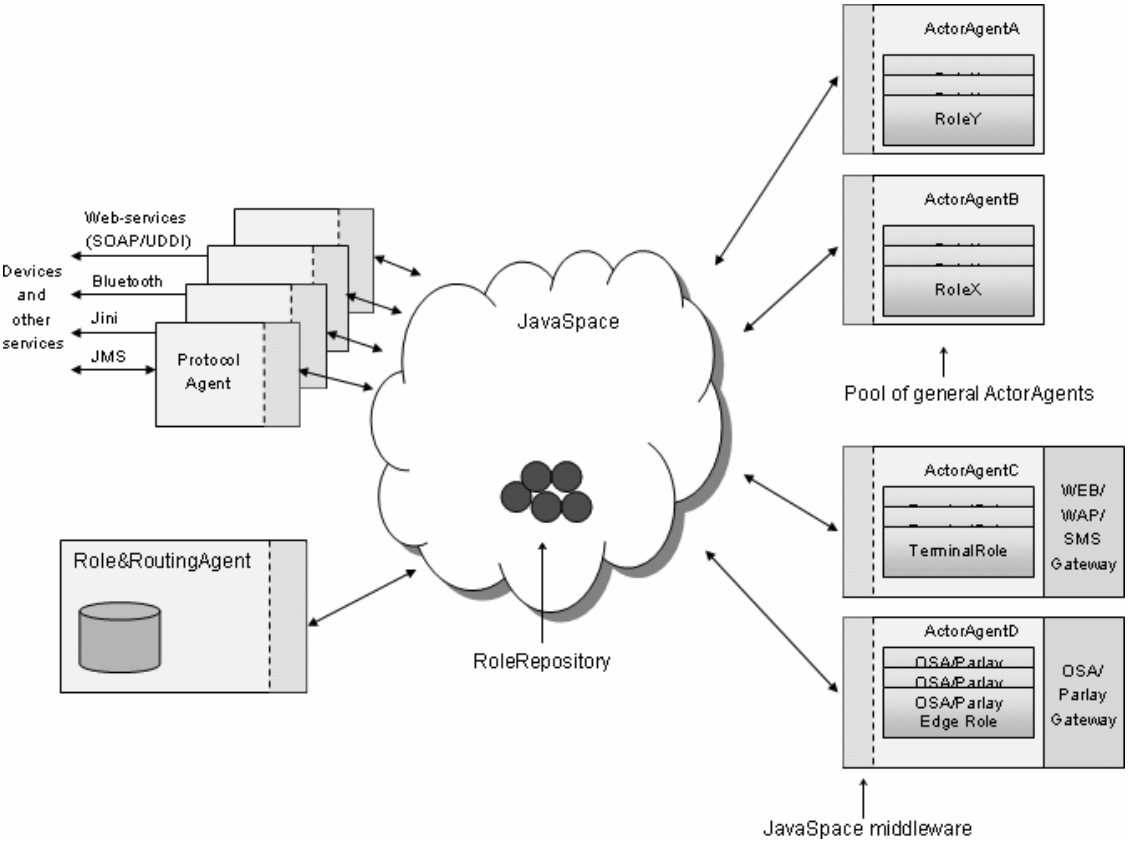

Figure 8.1 Conceptual space-based architecture

Agents
Only Agents are allowed to interact with a space. Three specialized types of Agents currently exist, ActorAgents, ProtocolAgents and Role&RoutingAgents.

ActorAgents are JavaFrame-based agents that are capable of playing different **roles** within a domain. A role represents specific behavior, and the concept is derived from ActorFrame. This is also the case with the Actor notion. The ActorAgent "containers" provide Jini and JavaSpaces middleware to the role "components" hosted. Some ActorAgents play specific roles, representing SMS gateways, web servers etc. ActorAgents capable of hosting more general service logic can exist in a pool, ready to do all sorts of work. We call these agents **Generic ActorAgents**.

Protocol Agents can do simple protocol translations of for instance the JMS, SOAP and Bluetooth protocols into the protocol that is used for interaction with the space. ProtocolAgents only do a simple mapping between protocols, and are not capable of playing roles. If more advanced protocol translation is needed, an ActorAgent playing a protocol translator role would be needed instead of Protocol Agents.

A **Role&RoutingAgent** knows which ActorAgents that are capable of playing the different roles, and thereby the capabilities of the domain. The Role&RoutingAgent is also responsible for the **Role Repository,** which is a logical collection of stored messages that contain role behavior (executable code) for the roles played in the domain. If the Role&RoutingAgent cannot find the proper role in its own domain, it may also forward messages to other domains (spaces).

Messages can either be addressed directly to the ActorAgent or they can be handled by an arbitrary ActorAgent that is qualified to process the Message. By qualified we mean that it has downloaded or is able to download a proper role from the Role Repository.

ActorAgents set up **event registrations** on the space in order to receive **notifications** when messages enter the space. After receiving a notification, the ActorAgent can determine to read, take or ignore the message from the JavaSpace. If more than one ActorAgent tries to take the same message from the JavaSpace, it is arbitrary which of the ActorAgents that succeeds. If this property is undesirable, messages have to be addressed directly to the correct ActorAgent by including a unique attribute that both the sender and the receiver know. A form of multicasting can be achieved if the ActorAgents read the messages instead of taking them.

8.3.2 An example scenario

In this section, we will go through an example of how the conceptual architecture could be used. The focus is on functionality, not technical details, which will be covered later. To describe the example scenario, three figures are presented.

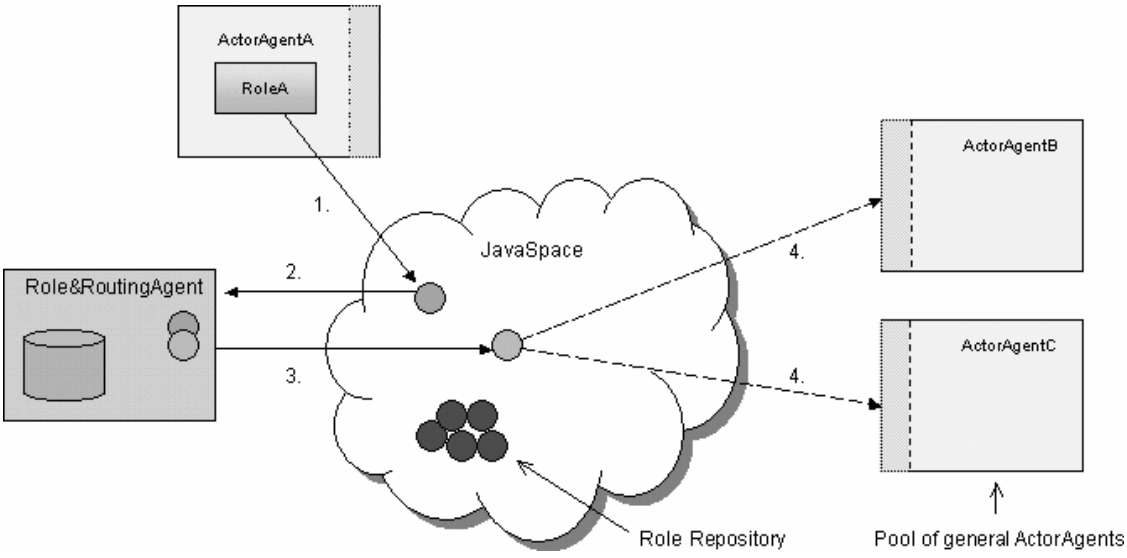

Figure 8.2 An example scenario, part 1

Figure 8.2 shows the first figure (part 1) of the example scenario. We will go through the incidents on the figure in numbered order. Starting with number 1, it represents a request included in an unresolved message that is written to the space. The sender, ActorAgentA has included its own identification (see section 8.3.7 for details) in the message but has not included any receiver address.

Incident number 2 represents the unresolved message being picked up by the Role&RoutingAgent, which finds out that the message can be handled in this domain. Incident number 3 illustrates how the Role&RoutingAgent then writes the message back to the space, this time as a resolved message, possibly indicating which ActorAgent(s) that can handle the message.

The two incidents that are represented by the number 4 illustrate two ActorAgents being notified about the resolved message that has been written to the space. In order for any ActorAgent to receive messages it has to set up a Jini remote event registration on the space. In this case, both ActorAgentB and ActorAgentC have registered templates that lead to a notification when a certain resolved message is written to the space. It is important to notice that the message itself has not yet been taken or read from the space. The decision whether to take, read or do nothing is made by the ActorAgents themselves.

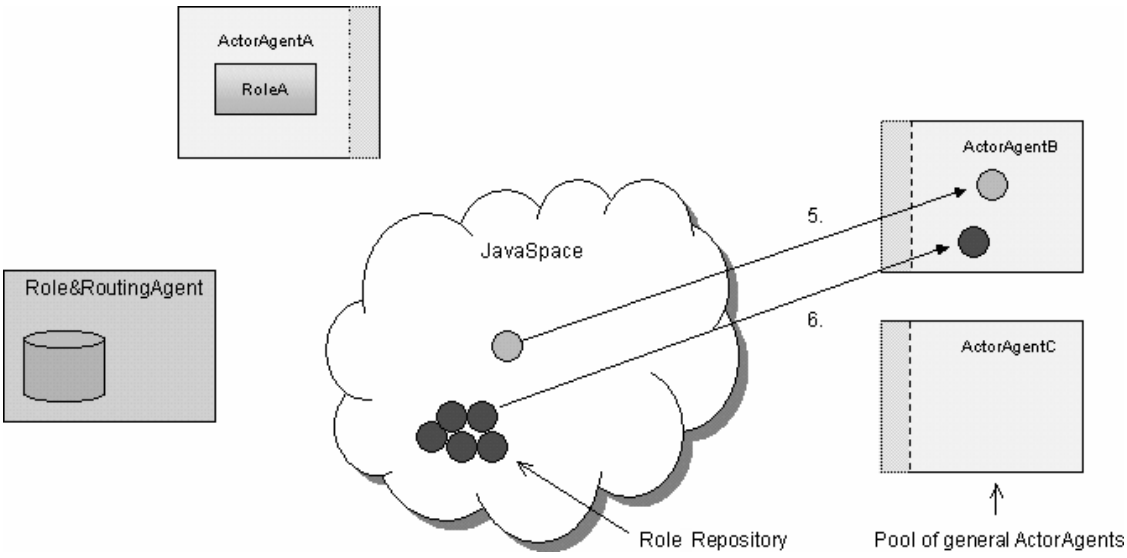

Figure 8.3 An example scenario, part 2

We continue with part two of the scenario (Figure 8.3), where incident number 5 shows that ActorAgentB takes the message from the space. In the example case, ActorAgentB does not yet know how to handle the message, so based on information in the resolved message it downloads a role that can provide this behavior. This is represented by incident number 6 in Figure 8.3.

After ActorAgentB has started to play the role that is required to handle the message (RoleB), it does the processing implemented in the role and returns a message to the space. This is represented by incident 7 in Figure 8.4 (part 3 of the example scenario). In this example case, the message is a session message, which includes the identification of ActorAgentA. ActorAgentA now gets a notification (represented by incident 8) about the arrival of a message with its identification. It can now read or take the session message that includes the result of its request. If ActorAgentB has included its identification, a session can now been established between ActorAgentA and ActorAgentB.

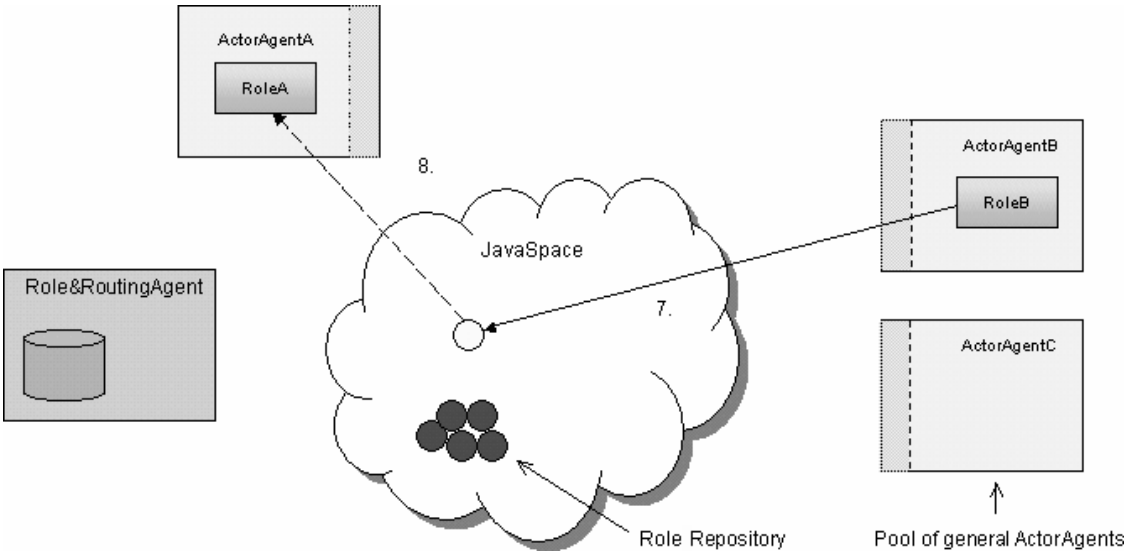

Figure 8.4 An example scenario, part 3

8.3.3 The ActorAgent

ActorAgents are much like the Actors in ActorFrame. Services are composed by combining different interacting ActorAgents that play different roles within the service. The reason why we call our actors for ActorAgents is that processes that interact by using the space communication paradigm often are called agents. ActorAgents contain specific behavior that enables them to exchange messages through a JavaSpace and receive role behavior from the JavaSpace. These properties should ultimately be the only ones that separate them from any other ActorFrame actors. In fact, our ActorAgents could be realized as a specialization of general ActorFrame actors, but for this project the main work has been done on the lower levels of adapting Jini and JavaSpaces principles.

In our conceptual architecture, ActorAgents are listening for messages that arrive in the space. This could be either messages they know they can handle or messages addressed directly to them. The ActorAgents are playing different roles, working together to deliver services. Like ActorFrame actors, ActorAgents are often a reflection of the environment that surrounds the system. As mentioned above, every part that wants to interact with the system has to be represented by an agent. Some ActorAgents are

representing specific entities in the "real world", like SMS gateways, WAP servers or Service Capability Servers (typically OSA/Parlay gateways, location servers etc. as explained in [1]) and therefore they are playing specific roles. Other ActorAgents are like "computation containers" capable of playing roles associated with more general service logic. These agents are referred to as Generic ActorAgents.

Sometimes it is necessary to address a message to a specific ActorAgent, for instance if the ActorAgent is playing a role that represents a specific SMS gateway. The case may also be that it is does not matter where or by who messages are handled, as long as an ActorAgent somewhere is able to play the appropriate role for handling the messages. Most of the messages will probably belong to sessions between roles in different ActorAgents, so it will be important that these messages are delivered to the owners of the session and not some other ActorAgent playing an identical role.

8.3.4 The Role&RoutingAgent

The Role&RoutingAgent is an Agent that picks up unresolved messages and analyzes their content. It then resolves where the messages can be handled. The Role&RoutingAgent knows which ActorAgents that are connected to the space and what roles they are able to play, thus knowing which roles are played in the domain. If no ActorAgents are able to play an appropriate role in the local domain/space, the Role&RoutingAgent should be able to know if and where to forward the message. Forwarding could be done to another domain/space where the message might be handled.

When a Role&RoutingAgent picks up an unresolved message and decides that it can be handled locally, it writes the message back to the space as a resolved message. This is done for two reasons. First, it is done to prevent the unresolved messages from being picked up more than once by a Role&RoutingAgent. Second, it is done to prevent the unresolved message from being read by all Agents listening to the space, which would be unnecessary and time-consuming.

A Role&RoutingAgent should also be able to instantiate new ActorAgents if it decides that the load on the existing ActorAgents is too heavy. The Role&RoutingAgent would then need to know about existing JVMs where it could start new instances of ActorAgents. Instantiation may be realized using the Jini start-libraries mentioned in chapter 3.2.7.

8.3.5 Messages

As we have seen, all objects that are written to a JavaSpace have to implement the tagging interface `net.jini.core.entry.Entry`. This implies that all JavaFrame messages have to be tagged with the `Entry` interface in order to be written to the JavaSpace. When composing a JavaFrame message, the `Message` class from the

JavaFrame package has to be extended. Consequently, all JavaFrame messages intended for communication through a JavaSpace implement the `Entry` interface.

A way to address messages is also needed, so that receivers can pick up messages from the space that are intended to them. The key point here is that the communication is based on associatively addressed messaging. JavaSpaces uses both object type and attributes to match the new entries written to the space with the registered templates. Clients use templates to describe which entries they want to get notified about, so that they can read or take them from the space (se chapter 3.3.3 on JavaSpaces). Our architecture operates with five basic massages for communication across a JavaSpace, but adding more message types is simple.

UnresolvedMsg
The UnresolvedMsg message is used initially, when the sender needs to interact with some other part but does not know where the other part is located or exactly which role is needed to handle the message.

ResolvedMsg
The ResolvedMsg message is used when the sender or the Role&RoutingAgent has determined where the message can be handled.

SessionMsg
The SessionMsg message is used to indicate that the message is part of an already established session with a specific peer and does not have to be picked up by anyone who is not a part of the actual session. When the SessionMsg is used, identifiers are needed as attributes so that the correct ActorAgents can get notified. See section 8.3.7 for a discussion of addressing and section 8.4.4 for comments on failure during a session.

RoleRepositoryMsg
RoleRepositoryMsg messages are used to store role behavior in the Role repository in JavaSpace.

CapabilityMsg
CapabilityMsg messages are used to communicate capabilities. This message is sent from ActorAgents to Role&RoutingAgents and might also be used between different Role&RoutingAgents to exchange information about different domains. The CapabilityMsg may also include an attribute to indicate to the Role&RoutingAgent whether new ActorAgents can be instantiated in the same JVM. As we will suggest in section 8.4.2, a CapabilityMsg could also be used for role trading.

Distinguishing between messages
There are two ways to create new messages: by subclassing the SpaceMessage class or simply by providing its type as an attribute in the SpaceMessage class. The two models

are shown in Figure 8.5 and Figure 8.6. There are several reasons to use an inheritance hierarchy, for instance the ability to use polymorphism in event registrations and having completely independent message structures. A combination of these two approaches could also be used, for instance a SessionMsg with an "importance" attribute set to "high".

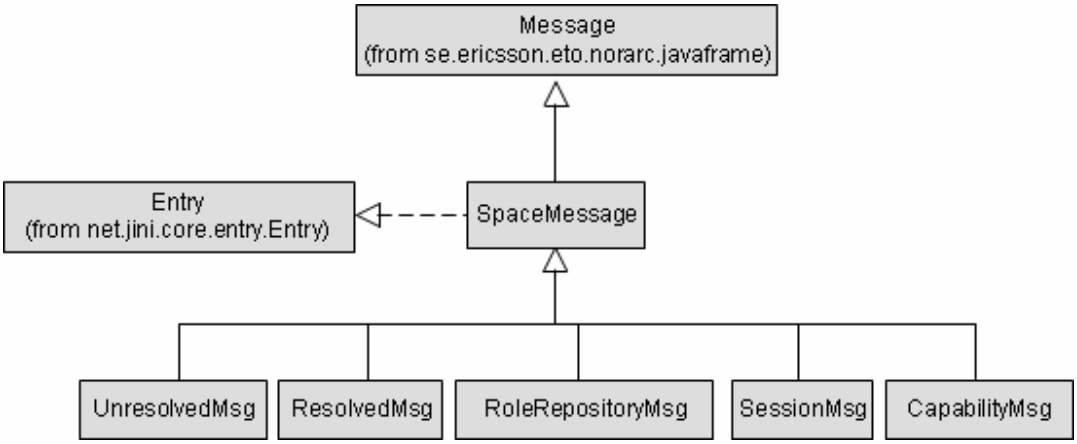

Figure 8.5 Using inheritance to distinguish between messages

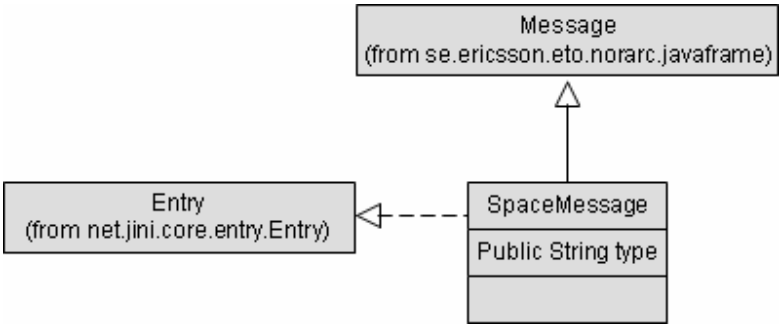

Figure 8.6 Using attributes to distinguish between messages

8.3.6 Interacting with JavaSpaces through SpaceMediators

In order to interact with a JavaSpace used as a Jini service, one has to go through the Jini discovery and lookup process described in chapter 3.2.3. If this process is successful, a Jini service proxy object implementing the `net.jini.space.JavaSpace` interface is returned. In the conceptual architecture, this functionality is embedded in two SpaceMediators, namely SpaceMediatorIn and SpaceMediatorOut. As the names suggest, one is for receiving messages from the space and one is for sending messages to the space.

Figure 8.7 is a simplified overview of how the SpaceMediators work. SpaceMediators can be placed on the edge of any JavaFrame-based system. An ActorAgent is just a specialized JavaFrame-based component, and this is the reason why Figure 8.7 does not show any ActorAgents. The figure also does not show how the asynchronous remote event notifications are used. The LUS shown in the figure represents the Jini Lookup Service, which the SpaceMediators use to find a JavaSpace.

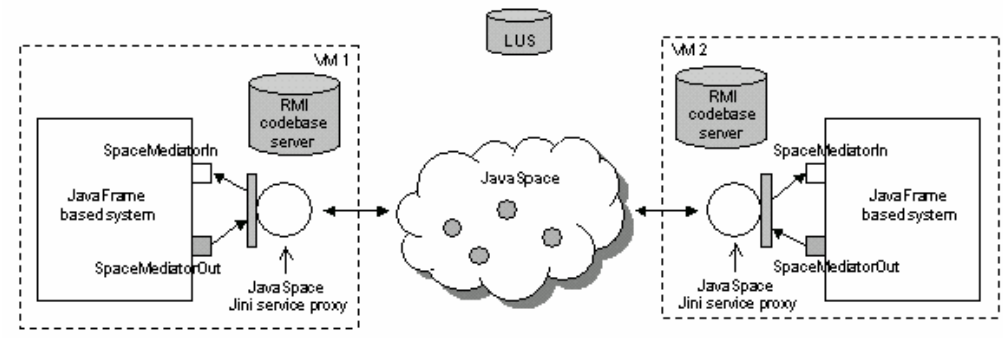

Figure 8.7 Jini service proxy and SpaceMediators

Compared to other mediators like RMI-mediators or CORBA-mediators that transparently deliver messages from one JVM to another, a more complex scheme of interaction is associated with mediators that support space-based communication. JavaSpaces work asynchronously in the sense that a write-operation on the space returns before the entry written to space is picked up by another process. By also using the asynchronous operations takeIfExists and readIfExists together with Jini's framework for asynchronous notification (remote events), we achieve totally asynchronous communication across the space. By using the asynchronous communication model, distributed ActorAgents can process messages at their own individual pace.

After receiving a remote event from the space, a readIfExist or takeIfExist operation must be performed to get hold of the actual entry that caused the notification. This model implies that there is no need for a single thread that waits for each type of incoming message or a single thread for each communicating peer. These properties fit well with the asynchronous message-based JavaFrame communication model, which is designed for minimal thread usage.

8.3.7 Inter-agent addressing

Addressing is an issue which can be solved by including one or more identification attributes in a message. The problem is: how can Agents obtain a (globally) unique ID when starting up? There are several ways to do this, but the Jini ServiceID, described in Table 3.1 Elements of a service item, will most probably be the basis for all solutions. The reason for this is that a Jini ServiceID already is globally unique, and therefore we already have a way of addressing a specific JavaSpace. We call a Jini ServiceID

belonging to a JavaSpace a SpaceID. The MAC-address could be used to identify the specific computer running the Agent in a JVM. To distinguish between different Agents in a single JVM, an AgentID could be added. The complete address would then look like Figure 8.8. Note that this is only an example of how the addressing problem could be solved, and that there might be better ways of doing this.

Figure 8.8 An example of a globally unique address

8.4 Advanced concepts for the space-based architecture

In this part, we present a few concepts that could further enhance the usefulness of the conceptual space-based architecture. The concepts are based on advanced JavaSpaces features and patterns.

8.4.1 Transactions

Starting with transactions, the Jini distributed transaction model presented in section 3.2.6 provides some interesting possibilities when used with a JavaSpace.

By using transactions, one can ensure consistent execution of specific actions. When participating in a transaction, objects in a JavaSpace will not be visible to Agents that are not participating in the transaction. Transactions will probably have many different uses in a space-based architecture. An example could be an ActorAgent connected to an SMS gateway and an ActorAgent representing a billing system. In such a scenario JavaSpaces' transaction capabilities could be used to ensure that either an SMS message is sent *and* the billing system is notified, or nothing is done.

Another case where it seems beneficial to use distributed transactions is when a generic ActorAgent downloads a role based on a message notification. Many things could go wrong in such a situation. For instance, the ActorAgent may not be able to obtain the correct role from the space. It would be preferable if the ActorAgent does not take the message before it is sure that the role can be played.

This problem may be solved by using a transaction on the RoleRepositoryMsg and the ResolvedMsg, which is shown in Figure 8.9. Now, the ActorAgent can either both read the role implementation *and* take the message, or do nothing. If it does nothing, the message will remain in the space, ready to be taken by any other agent that can play the necessary role.

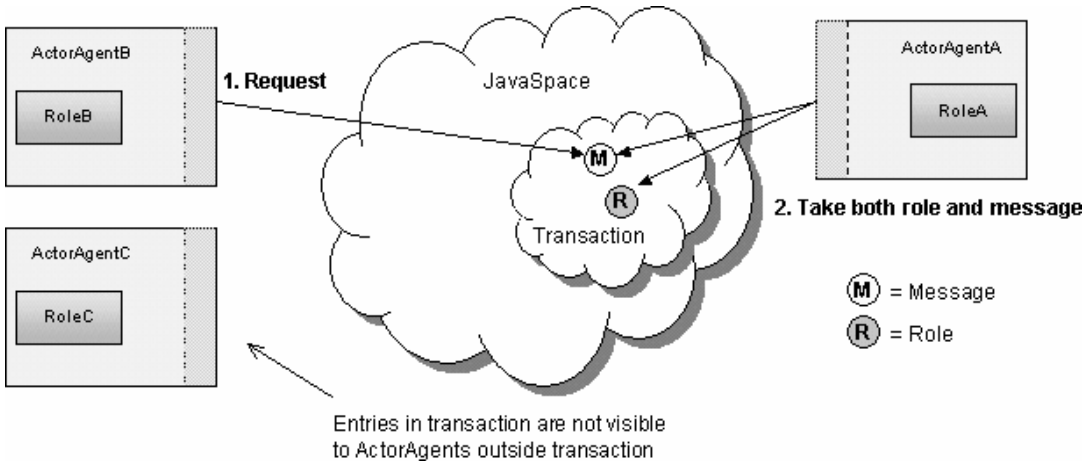

Figure 8.9 Using transactions when downloading a role

8.4.2 Trading of role capabilities

In [25], a pattern called the marketplace pattern is described. According to the book, "The marketplace pattern represents a framework in which producers and consumers of resources can interact with one another to find the best deal". This corresponds to trading as we know it from distributed systems theory, see [10] for more information about trading.

The marketplace pattern could be used in several settings in our space-based architecture, but the most interesting use is probably trading roles and capability of roles. Quality of Service (QoS) parameters could also be traded to ensure a maximum utilization of the system resources. Here, as well as with transactions, we will describe a scenario to illustrate the concept.

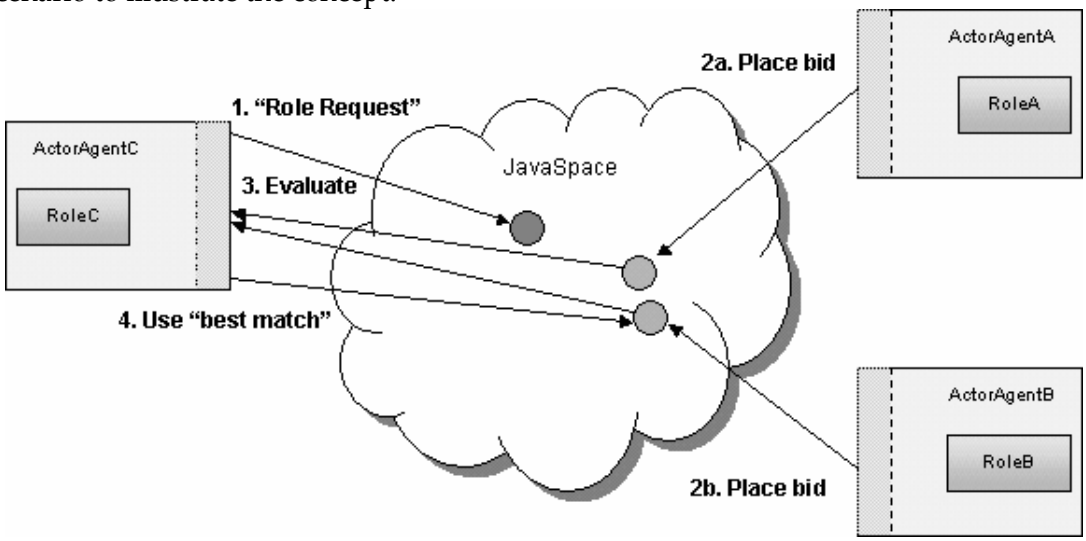

Figure 8.10 Trading roles

Incident 1 in Figure 8.10 shows ActorAgentC requesting a role, possibly with specific capabilities. The request may also contain interaction interfaces that it wants to use, providing Jini-like lookup functionality. ActorAgentA and ActorAgentB then answer what they are capable of (incidents 2a and 2b in Figure 8.10). It could be that they can not play the specific role that was requested, but can support a subset of the requested role behavior. ActorAgentA and ActorAgentC may "place bids", in terms of advertising their capabilities or QoS parameters, even though the capability is less than requested by ActorAgentC. The CapabilityMsg presented in section 8.3.5 could be used to distinguish these messages from other messages in the space. The message could include some kind identification as well, although not necessary.

Incident 3 represents ActorAgentC evaluating the "bids". This could be based on supported communication interfaces, capabilities or other parameters, such as QoS. If ActorAgentC finds that one of the "bids" is good enough, it can establish a session with the actual ActorAgent, based on its identification or simply by using its capabilities as an attribute. This is represented by incident 4 in Figure 8.10, where ActorAgentB is chosen. Taken to the extreme, ActorAgentA and ActorAgentB may collaborate to provide all the functionality (the role) needed. ActorAgentC may then make use of them both to realize its needs. This would probably require some interesting program logic at ActorAgentC.

8.4.3 Distributed data structures in JavaSpaces

Until this point, we have just presented the JavaSpace as a flat, unordered object store. It is also possible to organize objects in a tree structure or an array, for instance. Since many processes may access these data structures concurrently, they are called distributed data structures.

In [25], distributed data structures in a JavaSpace are explained as «…collections of objects that can be independently accessed and altered by remote processes in a concurrent manner». It is also explained that distributed data structures are hard to achieve in most distributed computing models, as «…these systems tend to barricade data structures behind one central manager process, and processes that want to perform work on the data structure must "wait in line" to ask the manager process to access or alter a piece of data on their behalf». JavaSpaces obviously is an elegant way of realizing distributed data structures, as there is no need for a central manager process.

In our conceptual architecture, distributed data structures could be used for organizing roles or messages. For instance, the Role repository could be organized in a tree structure to ease role identification, and messages could be organized in channels [25].

A channel in JavaSpaces terminology (shown in Figure 8.11) is a distributed data structure that organizes messages in a queue. Several processes can write messages to the end of the channel, and several processes can read or take messages from the

beginning of it. A channel is made up of two pointer objects, the head and the tail, which contain the numbers of the first and the last entry in the channel. It is possible to use several such channels, giving all ActorAgents associated with a space the possibility to handle messages in a FIFO fair manner. Channels may also be bounded, meaning that an upper limit can be set for how many messages a channel may contain.

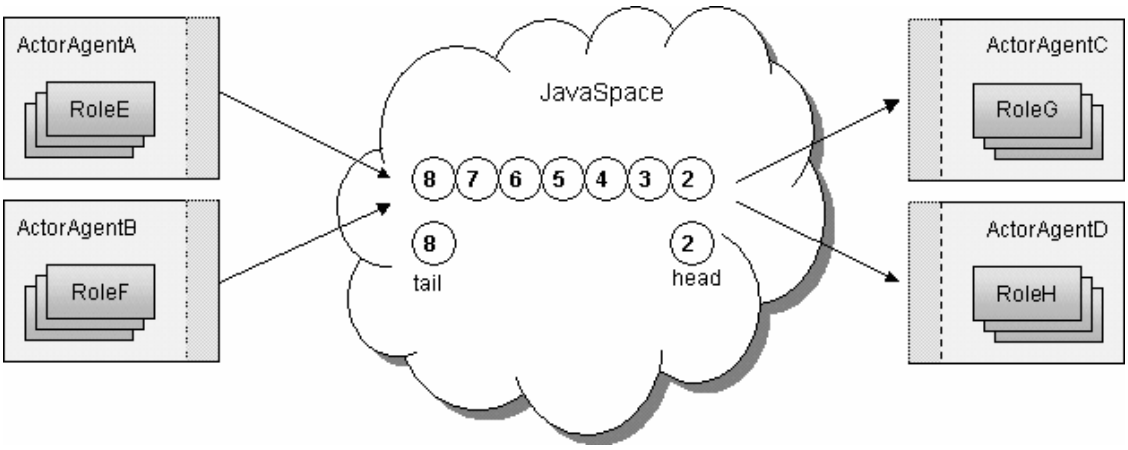

Figure 8.11 A channel

It is important to note that although messages are queued in a channel, it is still possible to use uncoupled communication as explained in section 3.3.2. The only thing that has to be agreed upon is the protocol used to realize the channel.

8.4.4 Using wills to ensure application consistency

ActorAgents are capable of establishing sessions across JavaSpaces by using addressed SessionMsg messages, but what would happen if an ActorAgent fails and a session is lost? This is a general problem associated with distributed computing and is not specific for space-based communication. Distributed leasing with Jini and JavaSpaces ensures that messages from a lost session will not stay in the space forever. Instead, they become garbage-collected after a certain time.

According to [45], «In a DSM system, although it is possible for the underlying infrastructure to detect that an agent has stopped responding, it is not easy for the agents to decide that another agent has failed. How this can be detected varies on the particular attributes of the DSM implementation. For example, in most tuple space based DSMs this is difficult/impossible to detect».

If an ActorAgent should fail during operation, the application state of the ActorAgent will most likely get lost. Transactions are used to ensure consistency in space-based systems, but this alone might sometimes not be enough to reconstruct application state and consistency in distributed data structures. In [45] it is suggested that an agent can

store its will in the distributed shared memory (space). If a space finds out that the agent has failed, its will is executed.

In the SpaceFrame architecture, ActorAgents could use wills to ensure system consistency. A will in SpaceFrame could be represented by a role with a special will behavior, just like the other role concepts. Should an ActorAgent fail, this role could probably be played by any generic ActorAgent, thereby executing the failed ActorAgent's "last wishes". The "last wish" could for instance be to notify other ActorAgents that the session is lost. In order to realize wills for JavaSpaces-based systems, extra functionality is probably needed.

Another approach to this problem could be to use some kind of timeout. For instance, if one of the communicating ActorAgents in a session has not responded within a certain amount of time, the session is considered lost. This solution is probably not as elegant as using wills.

8.4.5 Optimization through local spaces

In many cases it would be beneficial to have a number of ActorAgents running within one JVM. Sun's JavaSpaces specification states that RMI is to be used when interacting with a JavaSpace. Even if the ActorAgents were running within one JVM they would still need to use RMI for interaction with a JavaSpace. ActorAgents that communicate by using RMI and JavaSpaces introduce a substantial latency and need more system resources compared to communication within a single JVM's address room. A possible solution to this problem could be to use a "local" space implementation that is not dependent of RMI and may allow for optimized communication between locally deployed ActorAgents. This means that ActorAgents could be run within the same JVM as the space implementation. Doing this would combine the benefits of associatively addressed messaging with fast messaging within one JVM. The LighTS [46] open source project is lightweight Java implementation of a Linda tuple space that probably could be used to achieve this.

Figure 8.12 shows how this would look. A nice benefit of this approach is that it would be transparent to the interacting ActorAgents whether they are located within the same JVM or not. Messages would only leave the local JVM if they could not be handled by a local ActorAgent. Figure 8.12 includes a Routing Agent that would make the decision to forward messages that could not be handled locally. The Routing Agent would also forward messages from ActorAgents outside the local space to ActorAgents inside the local space. To achieve this, the Routing Agent would need to know which roles the ActorAgents in the local domain could play.

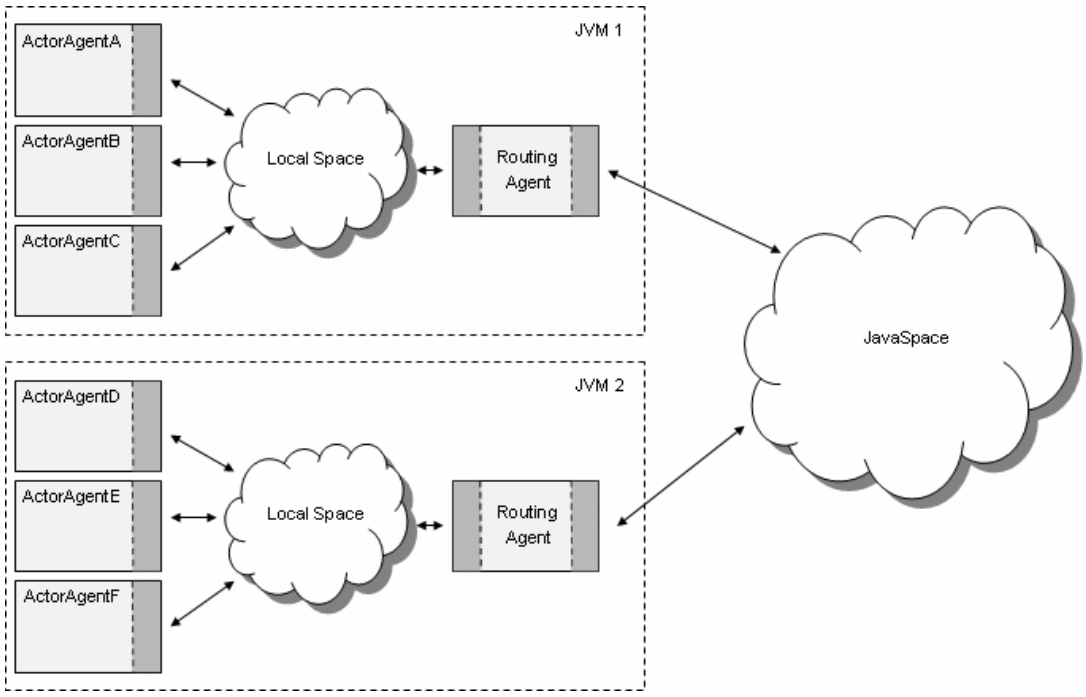

Figure 8.12 Using local spaces

8.4.6 Jini with a space-based architecture

Although there has been much focus on JavaSpaces in this chapter, it must not be forgotten that Jini provides all the infrastructure services for the conceptual architecture. For instance, the discovery of a JavaSpace is done entirely with the help of Jini. So let us take a look at Jini, and see if it can provide some additional functionality to such a conceptual architecture.

Downloading of ActorAgent software

When entering an environment where a space-based architecture is present, Jini could be used to download ActorAgent-software to computers that do not have any prior knowledge about the space-based system. As mentioned in chapter 7, Jini could even be used to download JavaFrame or any required software, and therefore would be ideal for use in an adaptive, dynamic environment. When an ActorAgent is started, it can download its role from the JavaSpace.

Start-library and life-cycle management

The Jini Start-library described in section 3.2.7 could provide some interesting functionality in a space-based architecture. The ActorAgents' container-approach could be further evolved with lifecycle-manager functionality. A management agent, whether it is a routing agent, a role agent or a load agent could monitor the space and start new instances on demand, as shown in Figure 8.13. This implies that a domain (space) not initially capable of handling a message could adapt to the situation, and start a new

instance that could play the role in question. A new instance could be started either because the role is not supported at the time, or because the load on an agent already playing the specific role is too heavy.

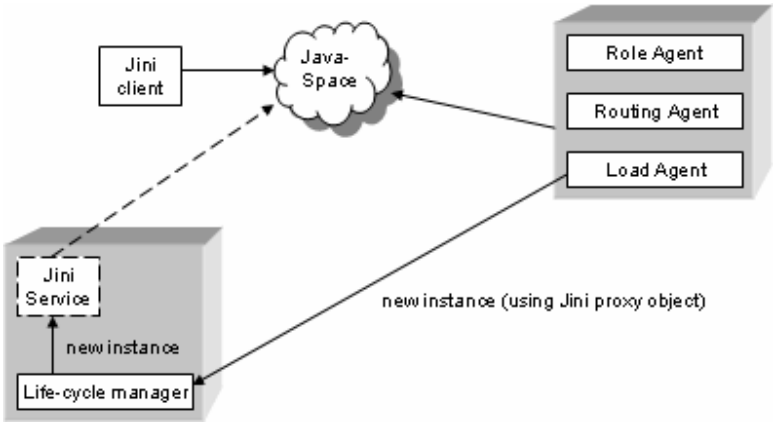

Figure 8.13 Life-cycle management using the Jini start-library

User interfaces

As mentioned in section 3.2.8, a specification has been made to standardize user interface (UI) downloading. A user interface, whether it is a graphical user interface or some other way of interacting with a user, can be obtained for a specific service and used as a layer between the user and the downloaded proxy object for the service.

The ServiceUI specification could be used when using Jini to download an ActorAgent to a terminal. Based on the capabilities of the terminal an appropriate user interface can be obtained. This means that regardless of display type, for instance, it would be possible to obtain a graphical user interface for a client-side ActorAgent.

It would be possible to place UI objects in the JavaSpace as well as role implementations. This "UI repository" would give the user of a space-based architecture the possibility to download a user interface for the role played. This would function in the same way as the Jini ServiceUI technology. The UI objects could be modules or plug-ins to the basic ActorAgent UI that was initially obtained by using Jini. It would probably be a good idea to use the JavaBeans component specification [47], with its introspection capabilities, to achieve this.

8.5 Summary

This chapter has described our conceptual space-based architecture on a theoretical level. The architecture builds on Jini's distributed event model, which we have used to achieve totally asynchronous communication through a JavaSpace. The conceptual architecture presented can prove to simplify distribution of JavaFrame-based systems in dynamic environments, partly due to JavaSpaces' powerful properties of uncoupled communication in time, space and destination. Our architecture introduces a simple and relatively untraditional way to distribute systems. By using such an architecture one can achieve location transparent communication between system components. We have introduced the concepts of a Role repository, downloading of roles on the fly and role trading, which are all powerful enablers for behavior distribution.

In the following chapter, we present the SpaceFrame framework prototype that has been made to illustrate the most important concepts presented above.

9 The SpaceFrame prototype framework

9.1 Introduction

The conceptual space-based architecture presented in the previous chapter introduces several ideas about everything from message distribution to role marketplaces. In this chapter we present our SpaceFrame prototype framework, which demonstrates the most fundamental concepts from our conceptual space-based architecture. By a framework we mean a set of Java class libraries and a set of rules that must be adhered to by the developer. This can greatly simplify the creation of JavaFrame-based systems that utilize JavaSpaces. The developer simply extends and specializes the framework to fulfill his or her needs.

We developed SpaceFrame to demonstrate how JavaFrame-based systems could benefit from JavaSpaces' ability to support loosely coupled, asynchronous communication and distribution of role behavior. By loosely coupled communication we mean uncoupling in time, space and destination, which are powerful enablers for communication in dynamic environments. As seen in chapter 2.1, JavaFrame's fundament is asynchronous communication. By using JavaSpaces in an asynchronous way, we can achieve a smooth integration with JavaFrame concepts.

9.2 What has been realized?

When developing the SpaceFrame prototype, we had to focus on implementing key functionality. With this functionality in place, it would later be possible to extend SpaceFrame's functionality. A full integration with ActorFrame/ServiceFrame, support for transactions and the marketplace pattern for role negotiation could then be implemented.

We have implemented a set of SpaceMediators that use Jini in order to obtain a JavaSpaces service proxy object. JavaSpaces and Jini's framework for remote events are used in a special way to ensure fully asynchronous communication across the JavaSpace. The SpaceMediators are associated with two sets of JavaFrame messages. One is used for communication through a JavaSpace, while the other is used for interaction between the ActorAgent and the SpaceMediators. The latter set of messages is used for the following purposes:

- Setting up and canceling event registrations on the space.
- Receiving notifications from the space as messages are written to the space.
- Reading or taking the correct message from the space.

We have also implemented a set of messages intended for communication between different JVMs across the JavaSpace. These messages enable associative addressing.

Furthermore, we have implemented a basic ActorAgent that uses the SpaceMediators for interaction with the JavaSpace. The ActorAgent contains special functionality associated with the SpaceMediators and the space-based communication paradigm. Once the ActorAgent receives an initial message, it requests an appropriate role from the Role repository in order to be able handle the received message. The prototype ActorAgent is only capable of playing one role at the time, but in the future it should be able to play many different roles concurrently, like the actors in ActorFrame.

9.3 ActorAgent implementation

As already stated, it should be possible to let ActorAgents be a specialization of ActorFrame actors, but for this prototype a minimal implementation is made in order to demonstrate key concepts.

ActorAgents can be seen as an advanced container-component approach where the container provides middleware for the hosted components. ActorAgents correspond to containers, while roles could be compared to components. As with actors in ActorFrame, ActorAgents are built with JavaFrame, from where they inherit the following properties:

- Composition
- Component independence
- Asynchronous communication
- State Machines
- Composite states

Figure 9.1 shows a common JavaFrame notation for visualizing active objects and mediators, including a notation for a JavaSpaces Jini service proxy object. In the future it should be possible for more than one ActorAgent to use the same pair of SpaceMediators, but this would require some kind of additional routing functionality in order to deliver the messages to the correct ActorAgent inside the specific JVM. It is possible that ActorFrame or the "local" spaces presented in section 8.4.5 could provide this routing functionality.

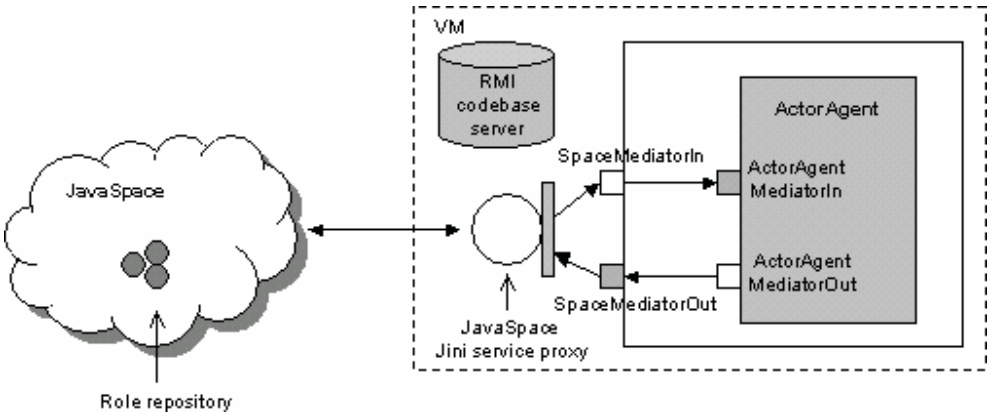

Figure 9.1 ActorAgent with mediators

Figure 9.2 shows a simplified class diagram for the ActorAgent. The ActorAgentCS (CS stands for composite state) contains functionality for handling messages that are used to set up an event registration on the space. The developer basically adapts the ActorAgents to suit his or her needs in two ways. First, the user of this framework puts his or her specific ActorAgent functionality in a class that extends ActorAgentCS. Then, specific role behavior is put in a class that extends SpaceRoleCS. Figure 9.2 shows that ActorAgentCS and SpaceRoleCS are declared abstract to ensure this.

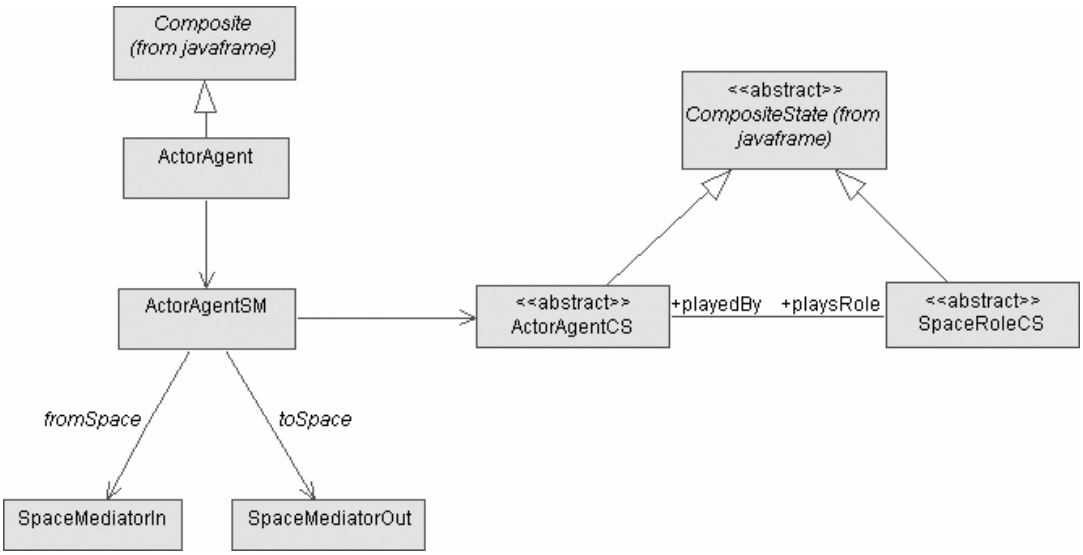

Figure 9.2 ActorAgent class diagram

Roles of type SpaceRoleCS are intended to be read from the Role repository in the JavaSpace and played by ActorAgents. All roles intended to be stored in the Role repository have to extend the SpaceRoleCS class.

In principle, this model allows any role of type SpaceRoleCS to be played by any ActorAgent. Sometimes, however, this may not be desirable. An example of this is if the ActorAgent is associated with specific capabilities, e.g. an SMS-gateway. This ActorAgent should probably only play roles associated with the SMS-gateway. Sometimes, security might also set restrictions on where roles can be played, but this is outside the scope of this thesis. ActorAgents probably should contain some additional functionality to ensure that only allowed roles are played. This is currently dealt with in two ways. First, ActorAgents can only take messages they know they can handle. Secondly, the fact that a Role&RoutingAgent knows about the ActorAgents capabilities also helps ensure that only allowed roles are played.

9.4 Role&RoutingAgent implementation

In addition to an ActorAgent, a simple Role&RoutingAgent has also been developed, purely to demonstrate communication principles. There are no means to dynamically discover the capabilities of the domain (space) as suggested in chapter 8. This functionality could probably easily be added later by extending the Role&RoutingAgent functionality that already exists. The implemented Role&RoutingAgent is shown in Figure 9.3. Notice that the Role&RoutingAgent utilizes the same type of mediators as the ActorAgent.

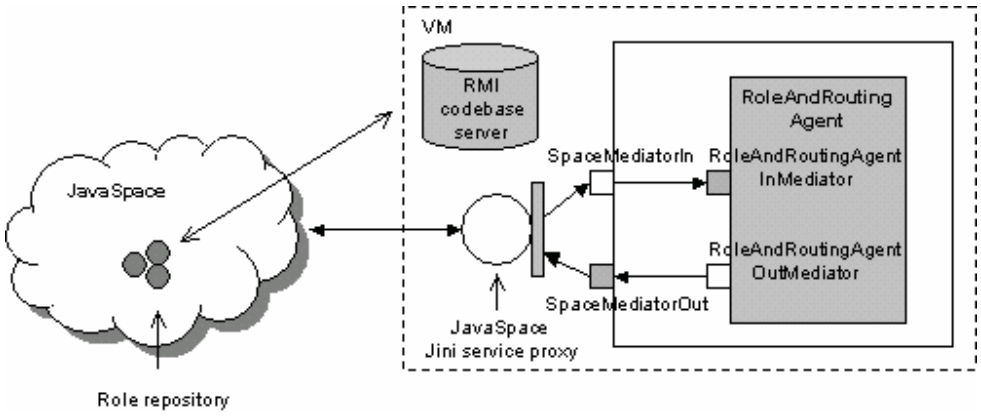

Figure 9.3 Role&RoutingAgent with mediators

The Role&RoutingAgent is listening for UnresolvedMsg messages that are written to the space. When it is notified of an UnresolvedMsg that has entered the space, it should pick it up and resolve where the message can be handled. The Role&RoutingAgent can do

this because it has knowledge about the roles that can be played in this and other domains.

Often, it is not important which ActorAgent who plays the required role. When this is true, the payload of the UnresolvedMsg message is written back to the space as a ResolvedMsg message without addressing any specific ActorAgent. However, the ResolvedMsg could contain specific attributes that would enable it to be picked up by ActorAgents with specific capabilities or resources. It is important to note that in many cases the ResolvedMsg message has to be addressed to a specific ActorAgent. This can be done by tagging the message with a specific identification that is only listened for by a single ActorAgent.

The SpaceFrame implementation of the Role&RoutingAgent is vastly simplified, as commented above. It simply wraps the content of the UnresolvedMsg message in a ResolvedMsg and writes it back to the space.

Figure 9.4 shows the JavaFrame state machine for the Role&RoutingAgent. First, the Role&RoutingAgent writes a RoleRepositoryMsg message to the space with a role that is described with a set of attributes. The Role&RoutingAgent also sets up an event registration for UnresolvedMsg messages with the SetupEventRegistrationMsg message. When the Role&RoutingAgent receives a NotificationMsg message, it sends out a TakeIfExistsMsg message, resulting in the SpaceMediators taking the UnresolvedMsg from the space. The UnresolvedMsg is then transported from the SpaceMediatorIn to the Role&RoutingAgent in an EntryFromSpaceMsg message. Now the Role&RoutingAgent simply writes the message back to the space, this time as a ResolvedMsg. It is important to notice that in principle, any agent may store a role in the space. In fact, any agent may store any kind of entry in the space, since all agents have access to the space at any time.

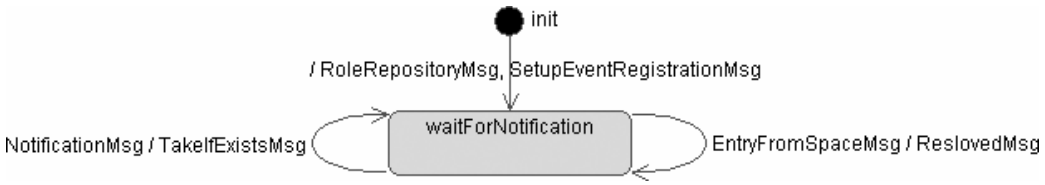

Figure 9.4 Role&RoutingAgent state machine

9.5 Messages

As we have seen in section 8.3.5, messages might be designed in many ways when using associative addressing. The messages we have designed can be separated in two classes:

- Messages for communication across a JavaSpace.
- Messages for interaction with SpaceMediators.

9.5.1 Messages for communication across a JavaSpace

In our prototype we have included four basic messages for communication across a JavaSpace (see Figure 9.5). These are a realization of the messages introduced in section 8.3.5. As can be seen, these messages share some common attributes. It might be a good idea to put these common attributes in a superclass that could be extended by all the messages, but for this prototype we have kept the design as simple and independent as possible. The designer decides if and how he or she wants to use the supplied attributes in order to make an addressing scheme.

It is important to notice that some kind of unique identification may be required to distinguish between ActorAgents. This identification is needed when it is necessary to address a specific ActorAgent and to identify a session between two ActorAgents. Our prototype framework does not provide this, so the user has to assign each ActorAgent its own unique identification. In the future, this process should be automated, providing each ActorAgent a unique identification. This identification could be generated in many different ways, and a suggestion is made in section 8.3.7 .

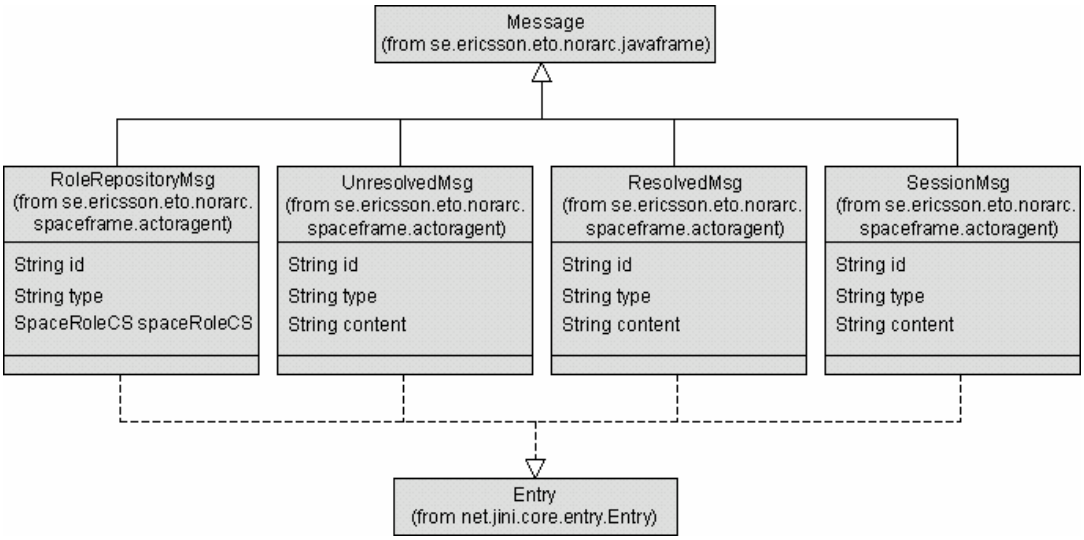

Figure 9.5 Messages for communication across JavaSpaces

Notice that the RoleRepositoryMsg in Figure 9.5 contains an object of type SpaceRoleCS. This object contains the role behavior to be played by an ActorAgent. Attributes on the RoleRepositoryMsg should be used in order for ActorAgents to find an appropriate role. A standard for describing the capabilities of roles should be made. This is especially important if the role trading concept presented in section 8.4.2 is to be used. The ActorAgents would need some way to compare the different roles in order to find out which roles that best suit their needs. How role behavior could be described is really a project in its own, so we have limited the SpaceFrame prototype to just using a simple string to describe what a role is capable of.

9.5.2 Messages for interaction with SpaceMediators

Along with the messages for communication across a JavaSpace, our framework also contains a set of messages used for interaction with the SpaceMediators (shown in Figure 9.6). These messages are used for controlling JavaSpaces-specific features, and have the following purposes:

SetupEventRegistrationMsg
A SetupEventRegistrationMsg is used to set up a remote event registration on the JavaSpace. The template attribute contains the template for the event registration, and will most likely be an UnresolvedMsg, a ResolvedMsg or a SessionMsg with various attributes. A lease attribute is an object of type `net.jini.core.lease.Lease` and is used to specify the preferred duration for a lease. The SpaceMediators contain a Jini lease renewal manager that renews the lease before it expires. In order for the ActorAgent to determine which event registration corresponds to which notification, the handbackObject is used. The same object that is included in the SetupEventRegistrationMsg message is returned upon event notification.

RemoveEventRegistrationMsg
A RemoveEventRegistrationMsg is used to remove a remote event registration on the JavaSpace. The template attribute is used to ensure that the correct event registration is removed.

NotificationMsg
When a message that matches one of the ActorAgent's event registrations enters the space, a NotificationMsg message is sent from SpaceMediatorIn to the ActorAgent. The NotificationMsg contains a remoteEvent object that is of type `net.jini.core.event.RemoteEvent`. The ActorAgent can find out which event registration that caused the notification by looking at the remoteEvent object, which contains the registered handbackObject.

TakeIfExistsMsg
The TakeIfExistsMsg is used to do a takeIfExists operation on the space with an entry included in the message as a template for the operation.

ReadIfExistsMsg
The ReadIfExistsMsg is used to do a readIfExists operation on the space with an entry included in the message as a template for the operation.

EntryFromSpaceMsg
The EntryFromSpaceMsg is used to transport messages from the space to the ActorAgent. The payload in the EntryFromSpaceMsg contains the actual message that has been taken or read from the space. The payload may be empty, that is, have the value `null` if no message matched the takeIfExists or readIfExists operation on the space.

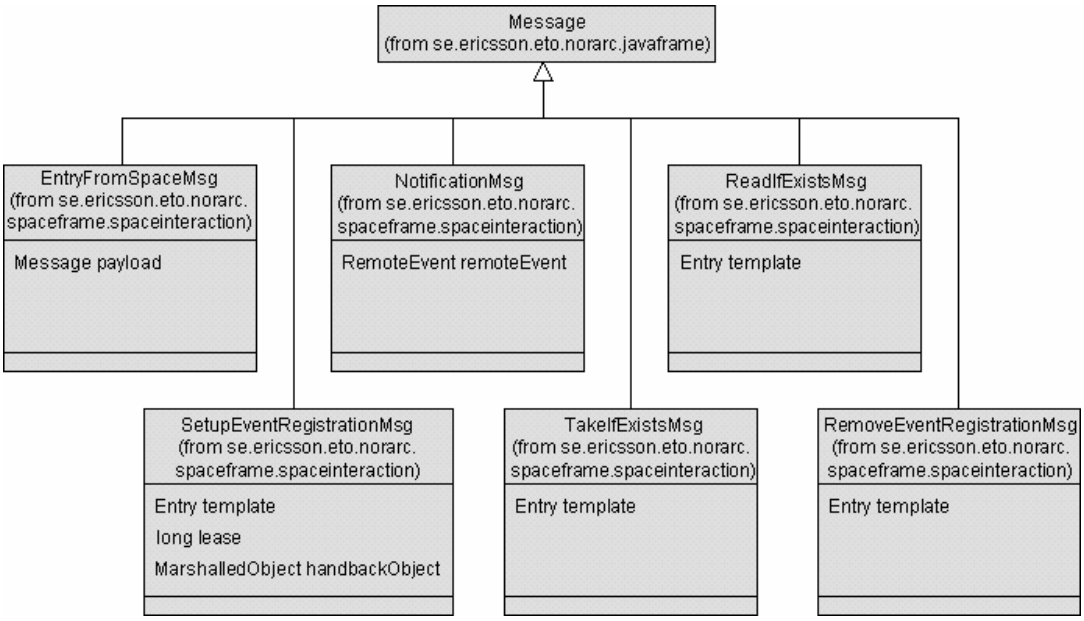

Figure 9.6 Messages for interaction with the SpaceMediators

9.6 SpaceMediators implementation

Due to time restrictions, we have tried to keep the implementation of the SpaceMediators as simple as possible, thereby leaving out some features that could prove to be useful.

The SpaceMediators simply use Jini to try to find a JavaSpaces service named "JavaSpaces". In many cases it could be useful to look up JavaSpaces with specific properties. ActorAgents in a service provider's domain would for instance look up specific JavaSpaces allocated for exchange of JavaFrame messages between ActorAgents that offer services. In the home domain, ActorAgents that participate in home automation would look for spaces with other properties. The point is that the SpaceMediators should be able to do more advanced discovery of spaces then they currently are capable of.

Another function that is left out is extensive control of Jini functionality. By Jini functionality we mean for instance control over the actions taken if a service should go down. For instance, the user of the SpaceMediators should be able to decide what to do if a JavaSpaces service becomes unavailable.

9.7 Roles and the RMI code base

When using RMI, objects are transported from one JVM to another by marshaling their content into a stream of bytes. The processes of turning objects into a stream of bytes and the byte stream back to objects are called marshalling and unmarshaling. Serialization is the Java equivalent of marshaling. In the process of unmarshaling an object, the JVM needs to know where to find the implementation (the class file) for the marshaled object. This information is embedded in the byte stream and is called a codebase annotation. The codebase annotation typically points to an RMI codebase server from where the unmarshaling JVM can download the appropriate class files.

The capability of RMI to unmarshal objects of classes that are not located within the local classpath of the executing JVM is especially useful when it comes to role distribution. In the prototype framework, this functionality is used in order for the ActorAgents to play roles whose implementations are not available in the local classpath.

The Role repository is a collection of RoleRepositoryMsg messages containing serialized roles. Class files for the serialized roles could in fact be located anywhere, but in our prototype implementation the role behavior is hosted by the Role&RoutingAgent's RMI codebase server. Figure 9.7 shows how the implementations of roles in the Role repository may be hosted by a number of different RMI codebase servers throughout the Internet.

Incident 1 in Figure 9.7 shows that the ActorAgent downloads the role object. Incident 2 shows how the ActorAgent's JVM contacts the correct RMI codebase server, indicated by a codebase annotation, in order to download the corresponding class file(s) for the role. Now the ActorAgent's JVM is ready to unmarshal the role.

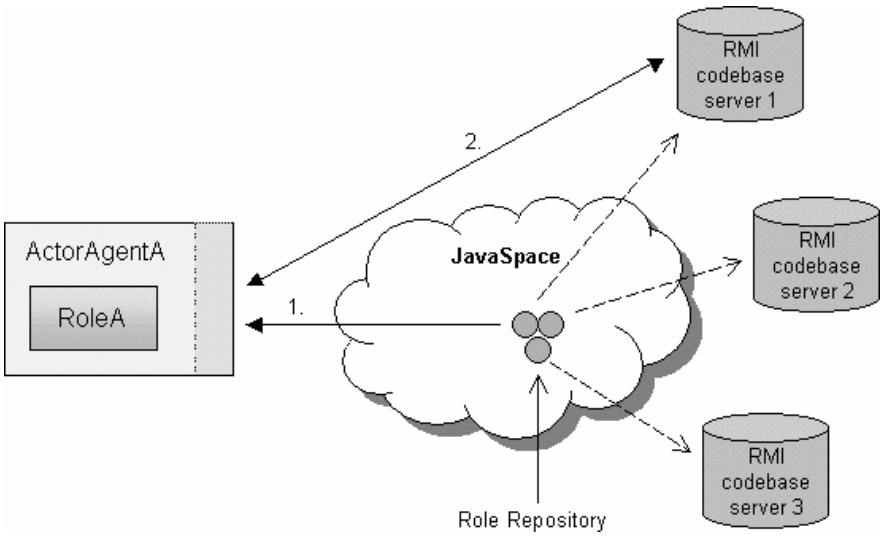

Figure 9.7 Role repository and the RMI codebase

9.8 Summary

The SpaceFrame prototype framework has been developed to show how the main concepts from the space-based architecture presented in chapter 8 can be realized. We have implemented a set of SpaceMediators, an ActorAgent, a simple Role&RoutingAgent and two sets of messages. The messages are used both for controlling space-specific functionality in the SpaceMediators and for communication across a JavaSpace. Our work with the SpaceFrame prototype has shown that using JavaSpaces is an elegant way to interact asynchronously in a dynamic JavaFrame-based environment. To prove the usefulness of the SpaceFrame framework, we have developed an example application that utilizes SpaceFrame to make a JavaSpaces-based system. This example application is described in the next chapter.

10 Using SpaceFrame

10.1 Introduction

In order to demonstrate how our conceptual space-based architecture works, and to show how SpaceFrame can be used, we have developed an example application. This application basically demonstrates how JavaSpaces as a Jini service can be used for role distribution and as advanced middleware for interaction. The example application is very simplified user "authentication" system, and utilizes uncoupled communication, which is a powerful feature in dynamic environments.

To build the example application, we have extended the SpaceFrame prototype. This has been done by adding a specialized ActorAgent, a specialized Role&RoutingAgent, a TiniIbuttonAgent functioning as a ProtocolAgent and a TINI Jini node. TINI stands for Tini InterNet Interface and is a Java programmable embedded Ethernet platform. The "iButton" is a small computer chip in a steel can that can host everything from a Java VM to different kinds of environmental sensors. More information on TINI and iButton is available from [48].

10.2 Deployment view

Figure 10.1 shows a simplified UML deployment diagram for the example application. The boxes marked gray are the most important entities in the example application. It is worth noticing that not all Jini *service interaction* in the deployment diagram is based on RMI. The TINI node's service proxy object uses PM-RMI (Poor Man's-RMI), a lightweight RMI implementation, to interact with the TINI. This is an example of Jini's ability to use different protocols between the proxy object and the service. Lightweight Jini functionality called "Jini for small places" is used on the TINI module. Both "Jini for small places" and PM-RMI are developed as a part of the Anhinga [49] project.

The deployment diagram also shows a number of standard Jini processes, including the RMI activation daemon, an RMI codebase server and the Jini Lookup Service (LUS). We have already treated the RMI codebase and the Jini Lookup Service (LUS). The RMI activation daemon is a standard part of the JDK (Java Development Kit) which is used to start activatable objects.

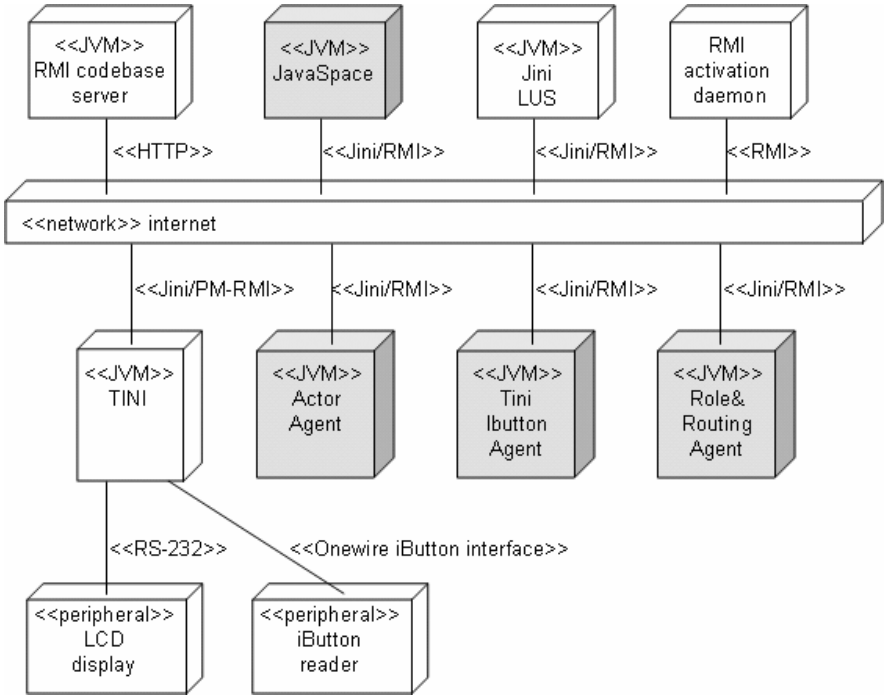

Figure 10.1 UML deployment diagram for example application

10.3 TiniIbuttonAgent and TINI Jini node

The example application is a very simplified user "authentication" system. As shown in Figure 10.2, a TINI (2) is connected to an LCD-display (1) and an iButton reader (4) that can read data from iButtons (3). Jini is used to connect the TINI with a TiniIbuttonAgent that is capable of interacting with a JavaSpace. This is shown in Figure 10.3, which also shows that the TiniIbuttonAgent uses Jini to find the JavaSpace, meaning that Jini is used twice to enable communication from the TINI to the JavaSpace. An ActorAgent and a Role&RoutingAgent are also connected to the same JavaSpace. Detailed figures of the ActorAgent and the Role&RoutingAgent are shown in sections 8.3.3 and 8.3.4.

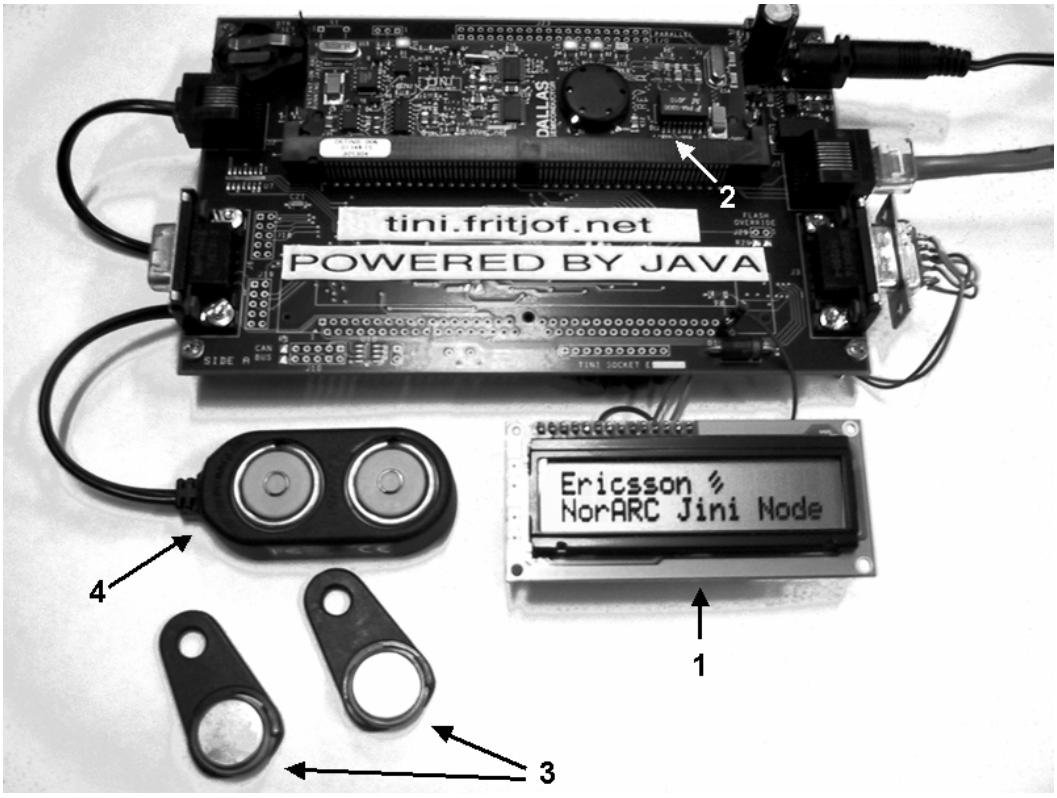

Figure 10.2 TINI Jini node with iButton reader and LCD

The TiniIbuttonAgent registers a remote event listener on the iButton reader through a Jini service proxy object for the service hosted on the TINI. When the reader detects an iButton, an identification string is sent inside a remote event to the TiniIbuttonAgent. The TiniIbuttonAgent does not know where and by whom this identification string can be resolved. It therefore writes an UnresolvedMsg message to the JavaSpace, containing the iButton identification string, the "type" attribute "IBUTTON_EVENT" and its own identification.

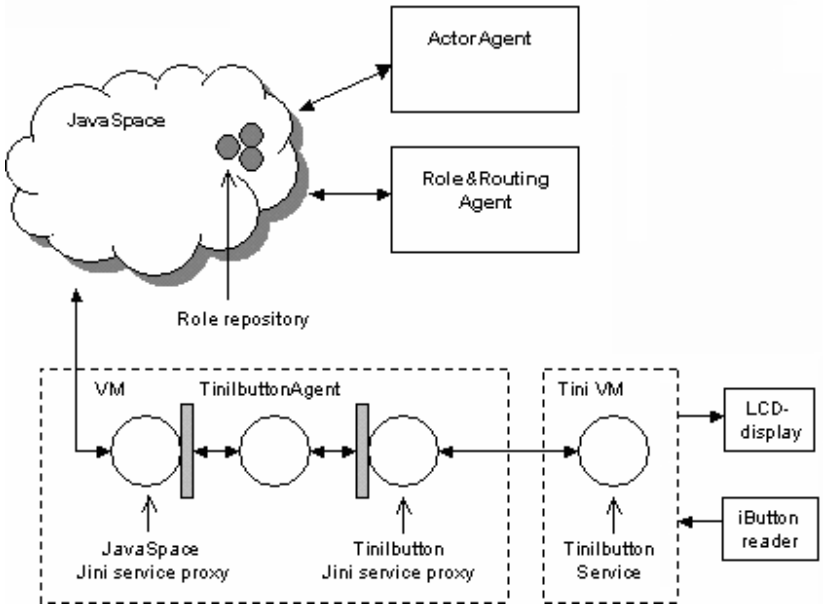

Figure 10.3 Example application

10.4 Role&RoutingAgent

The Role&RoutingAgent in the example application system is a customized version of the SpaceFrame Role&RoutingAgent implementation. When starting up, the Role&RoutingAgent writes a RoleRepositoryMsg message to the space with a role that is capable of handling an "IBUTTON_EVENT". This means that it knows that a generic ActorAgent connected to this JavaSpace is capable of playing a role that can handle messages with the "IBUTTON_EVENT" attribute.

When notified of an UnresolvedMsg, the Role&RoutingAgent takes the message and finds that it can be handled in the local domain (space). Since it is not important who handles the message (plays the role), the payload of the UnresolvedMsg message is sent back into space as a ResolvedMsg message without any specific addressing. The payload consists of an "IBUTTON_EVENT" string, the TiniIbuttonAgent's identification and the iButton's serial identification string.

10.5 ActorAgent

When the Role&RoutingAgent writes the ResolvedMsg message to the space, the generic ActorAgents get notified with NotificationMsg messages. The NotificationMsg messages result in the ActorAgents trying to take the message corresponding to their event registration. Only one of the ActorAgents will succeed in taking the ResolvedMsg message from the space.

After getting hold of the ResolvedMsg message, the ActorAgent realizes that it does not know the exact role behavior in order to handle the incoming message. This results in the ActorAgent saving the message and contacting the Role repository in order to obtain an appropriate role. The ActorAgent does not know which role is capable of handling the message so it simply asks the Role repository for a role that can handle an "IBUTTON_EVENT". To achieve this, the ActorAgent instructs the SpaceMediators to do a read-operation on the space with a template RoleRepositoryMsg message that has the type attribute set to "IBUTTON_EVENT".

The example application builds on the SpaceFrame prototype described in the previous chapter. In addition, a GenericAgentCS and an IbuttonRoleCS are added. This can be seen by comparing Figure 9.2 with Figure 10.4.

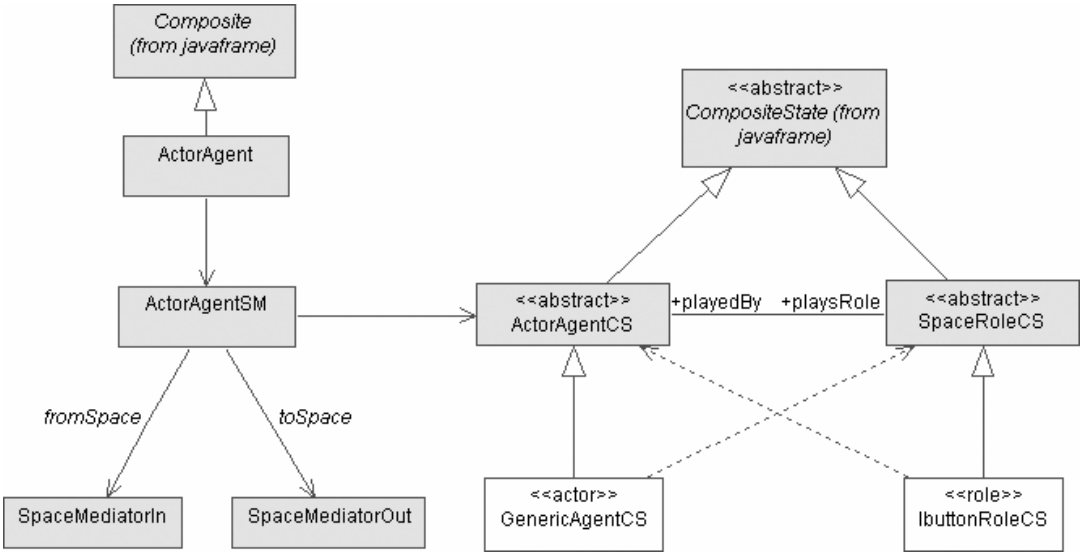

Figure 10.4 Class diagram for example application's ActorAgent

A SpaceRoleCS is included in the RoleRepositoryMsg message, containing the actual role behavior for handling the message (see Figure 10.5). The SpaceRoleCS is used to define the inner behavior of the ActorAgent's playRole state. So when the ActorAgent has plugged in the received RoleRepositoryMsg message to define the behavior of its inner state machine, it is ready to handle the saved ResolvedMsg.

Figure 10.5 shows the ActorAgent's state machine for this process. A slightly modified version of the UML standard is used to show the inner state of the ActorAgent's playRole state. As can be seen in Figure 10.5, the playRole CompositeState has an entry port and an exit port, which are both marked "1". These are used when entering and exiting the composite state. More entry/exit ports may be used if there are more ways to enter/exit the CompositeState.

In the example application an IbuttonRoleCS is capable of handling multiple ResolvedMsg messages with the attribute "IBUTTON_EVENT". The IbuttonRoleCS contains the behavior that makes the ActorAgent look for an iButton identification string inside the ResolvedMsg message. This string is used to find the user name of the iButton owner, which is then sent back across the space in a SessionMsg message addressed specifically to the TiniIbuttonAgent. When the TiniIbuttonAgent receives the SessionMsg message, it calls a method in the TINI's Jini service proxy in order to display the user name at the LCD.

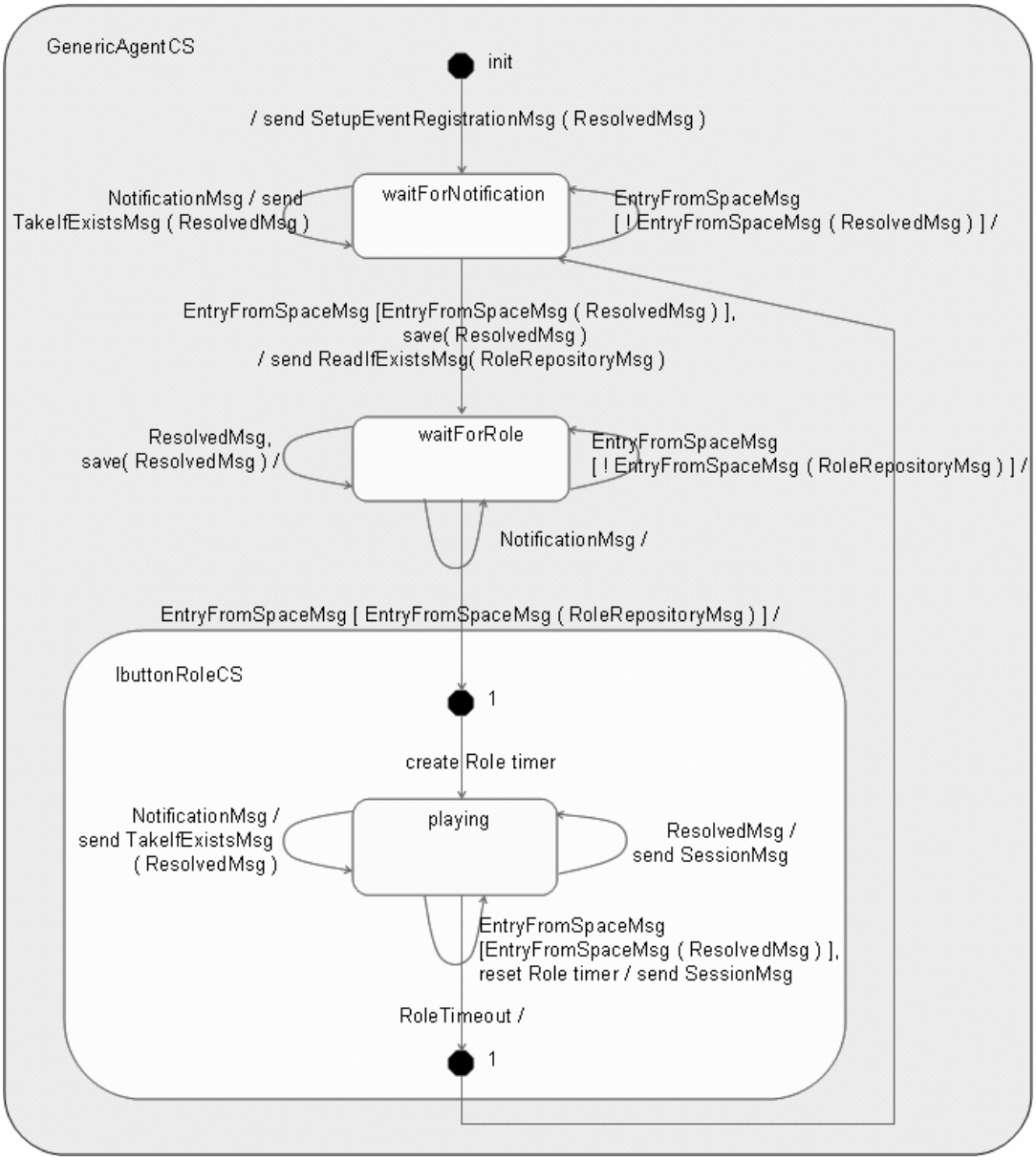

Figure 10.5 State machine for the example application's ActorAgent

It is also worth mentioning that the IbuttonRoleCS composite state contains a "role timer". When ActorAgents exist in a pool it might be a good idea if some ActorAgents stop playing their roles if their behavior has not been requested for some time. This would free the ActorAgents to start playing new roles. This is why a role-timer is introduced in the example application. If the ActorAgent that is playing the role associated with IbuttonRoleCS has not received a ResolvedMsg message in a given amount of time, the ActorAgent exits its playRole state and thereby becomes ready to play a new role. This would allow the system adapt to new needs in the domain.

10.6 Simplified sequence chart for example application

Figure 10.6 shows a simplified UML sequence chart for the example application. The sequence chart shows the message flow that corresponds to a success scenario like the one described earlier in this chapter. It is important to notice that not all messages sent to the "JavaSpace" instance in the sequence chart actually get stored in the JavaSpaces distributed object store. Some messages are used for controlling the SpaceMediators. These messages are SetupEventRegistrationMsg, TakeIfExistsMsg and ReadIfExistsMsg. A non-standard UML notation is used to describe that asynchronous messages contain inner messages and roles, e.g. that a RoleRepositoryMsg contains an IbuttonRoleCS and that an EntryFromSpaceMsg contains an UnResolvedMsg.

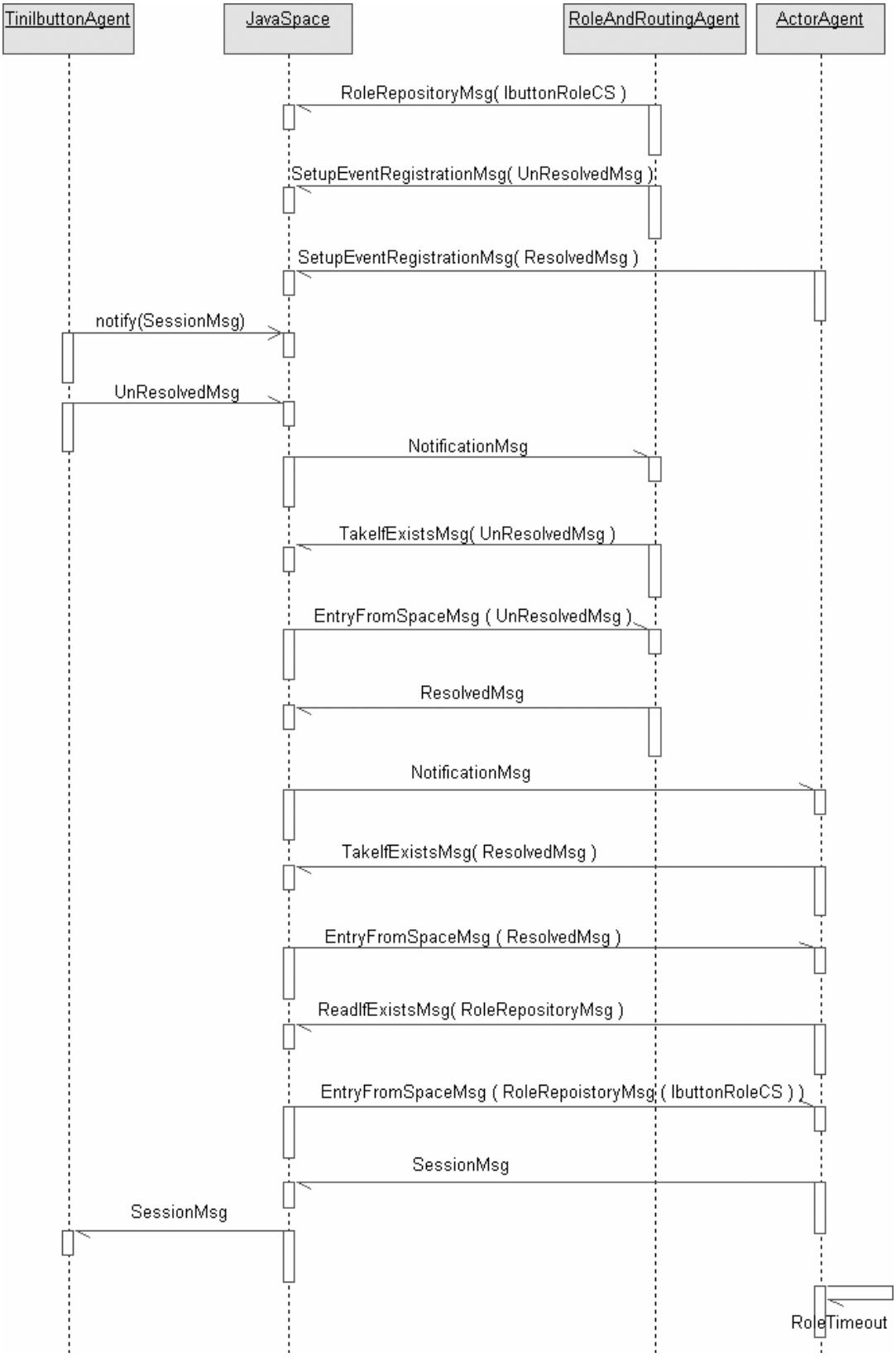

Figure 10.6 Example application sequence chart

10.7 Summary

The example application described in this chapter can be seen as evidence that the foundation of our conceptual space-based architecture can be realized. End-to-end asynchronous communication is used, providing a smooth integration with the JavaFrame world. Various forms of uncoupled communication are used, including both direct addressed communication and unaddressed communication. Dynamic role downloading is also demonstrated, providing an elegant solution for behavior distribution. We have also shown how a developer can relatively easy develop a JavaSpaces-based system by using our SpaceFrame framework. In the following chapter we will discuss our results before giving a conclusion in chapter 11.

11 Discussion

11.1 Introduction

In this thesis, we started out by performing a brief evaluation of three frameworks building on Jini and JavaSpaces. We carried on with evaluating how Jini could be used together with JavaFrame-based systems, before presenting our own space-based architecture. We then described our SpaceFrame prototype, which implements a subset of our space-based architecture. Finally, we demonstrated how SpaceFrame can be used by presenting an example application.

The purpose of our work has been to see if Jini, JavaSpaces and frameworks building on these technologies can be used to adapt JavaFrame-based technologies to a dynamic environment. In such an environment, clients and services come and go all the time and system components may dynamically be added and removed. When using Jini and JavaSpaces with JavaFrame-based systems, a service-oriented programming model must be used, which means that systems must be composed of logical components that offer and/or use services. This approach might force developers to think in a new way when designing systems.

This discussion is structured in the same way as the main part of the report. The following issues are discussed:

- The three frameworks that build on Jini and JavaSpaces: Rio, Openwings and SI.
- Using Jini with JavaFrame-based technologies.
- Using JavaSpaces with JavaFrame-based technologies, a conceptual architecture.
- The SpaceFrame prototype. This point also includes a brief discussion of the example application developed.
- Further work that could be done in this area.

11.2 Rio, Openwings and SI

Rio, Openwings and SI are frameworks that build on Jini and JavaSpaces. These are interesting technologies, but they are all quite immature, and especially the SI and Rio architectures are sparsely documented. We have therefore focused on the key concepts that they represent.

As mentioned in chapter 5.6, we have found the Rio architecture to be a subset of Openwings. We also think Rio's dependency on Jini could be seen as a weakness when compared to Openwings, which aims at abstracting over different service discovery mechanisms.

In Openwings, an architecture description language is used to model the interaction between services, with the notion of a connector being a central concept. Openwings' asynchronous connectors would probably fit well with the asynchronous JavaFrame technology and its mediators. The Openwings component model also does not place restrictions on the threading model used in Openwings components. Therefore, it should be fully possible to deploy JavaFrame-based components inside Openwings containers, including both ActorFrame Actors and SpaceFrame ActorAgents. This would allow JavaFrame-based components to benefit from advanced container services, including deployment, clustering, fail-over and security features.

A weak point with Openwings could be that it is a quite comprehensive framework that may place undesirably high demands on the computers running it. Compared to ServiceFrame, which is very compact and effective, this could be an unacceptable limitation.

SI is the least documented of the three frameworks, and the current release is immature. It is therefore difficult to do a thorough evaluation of how this architecture could be used together with JavaFrame-based technologies. However, it is clear that SI tries to solve common problems with space-based architectures, and that asynchronous communication is possible. SI may well be a technology to observe as it evolves.

As the conceptual space-based architecture we have developed is influenced by SI, it makes sense to ask why SI could not be used instead. As mentioned above, SI is quite immature and poorly documented, so we found that we could just as well implement our own system and architecture, customized for use with JavaFrame and its relatives. There is, of course, also the learning perspective of developing such a system from the ground up.

11.3 Using Jini with JavaFrame based technologies

Jini, with its downloadable proxy object model, is designed to be used in a dynamic, context-sensitive environment, where services may become unavailable at any time. There are many benefits that can be drawn from using Jini, the most important being protocol independence and adaptivity. A client can find and use services by knowing the Java communication interface and its semantics, and possibly some attributes.

Developing a set of Jini mediators that are capable of doing service discovery could prove to be a flexible way to let JavaFrame-based components form into systems at runtime. In fact, more code than just the protocol implementation may be downloaded. A client may for instance download an entire JavaFrame runtime environment embedded in the Jini proxy object. The client would not even need to know anything about ActorFrame and ServiceFrame concepts such as actors and roles, which transparently may be distributed to the client.

11.4 Using JavaSpaces with JavaFrame-based technologies, a conceptual architecture

JavaSpaces technology provides several powerful features for use in advanced distributed applications. For instance, it is possible to use loosely coupled communication in time, space and destination. In addition to other valuable properties like scalability, using a space also ensures that as long there exists a service that can respond to a message, it will do so. Several mission-critical applications have been built on the tuple space paradigm.

The JavaSpaces notify method provides an asynchronous way of messaging, built on the Jini distributed event specification. Asynchronous communication can be used end-to-end without extensive use of threads. As asynchronous communication is a key concept in all JavaFrame-based systems, using JavaSpaces with these systems will most probably be a good choice.

We have developed a conceptual space-based architecture to provide JavaSpaces functionality to the JavaFrame environment. The conceptual architecture is based on distributed events and message passing via JavaSpaces, leading to asynchronous communication end-to-end and minimal thread use. This fits well with the JavaFrame philosophy.

When building this system, we have taken the distribution problem into account right from the start. We have introduced a new concept called a Role Repository that is a powerful way of using JavaSpaces for behavior distribution. With the Role Repository, role behavior can be downloaded from a space on the fly by generic agents. This allows for an adaptive system to be made, with ActorAgents changing the roles they play based on the current demands on the system. Not all agents can do this, for instance agents running on computers that have specific hardware installed or specific permissions granted should not be generic agents. Note that due to the small size of JavaFrame and SpaceFrame it is fully possible to run ActorAgents on some of the more powerful mobile terminals. However, current J2ME [50] profiles do not support RMI and Jini, which are needed to run ActorAgents. In the future, a J2ME profile supporting Jini could even make it possible to run ActorAgents on mobile phones.

Several interesting concepts have emerged from our work with the space-based architecture, including role trading and using transactions to ensure correct handling of messages. Jini also adds some functionality to the space-based architecture in addition to providing infrastructure mechanisms, including instantiation functionality and a way to globally identify a JavaSpace. It is, however, important to notice that the space-based architecture is conceptual and must be seen as a platform for a possibly evolving system.

11.5 Prototype and example application

The SpaceFrame prototype implementation has been made to prove that it is possible to realize our space-based architecture. An example of this would be the asynchronous communication principle, which is now extended to the space, as we have used the asynchronous notify method from the JavaSpace interface.

We have developed a set of SpaceMediators that help dealing with JavaSpaces-specific functionality. When developing a space-based framework based on JavaFrame, an issue which arose was the use of templates for describing which messages to be notified about. This functionality had to be modeled within its own state machine, passing messages to a SpaceMediator from inside the system. Therefore, the developer has to be aware that he or she is using SpaceMediators, contrary to most other mediators. When designing a system based on the SpaceFrame prototype framework, these mediators have to be used and the ActorAgent state machine has to be extended. This provides a fairly simple way for a developer to make JavaFrame-based programs able to communicate through JavaSpaces.

The result of our work with SpaceFrame is a functioning prototype that can be used in dynamic environments, with communication that can be uncoupled in time, space and destination. In addition, we have demonstrated dynamic downloading of a role, which times out and must be downloaded again after the timeout. This provides a developer with flexibility, and can be seen as evidence that our space-based architecture is possible to implement. Much of the advanced functionality, however, has been left out, due to time restrictions. The Role&RoutingAgent functionality in the prototype is not properly implemented, and therefore the SpaceFrame prototype can only be used with one space in its present state.

In the current version of the SpaceFrame prototype, the SpaceMediators can not be used to control Jini-specific functionality, for instance how to handle a failed JavaSpace. All the functionality related to Jini is embedded in the SpaceMediators. Ideally, the SpaceMediators should be using some JiniMediators to discover a JavaSpace. Again, time has been the reason why this has not been done in the first version of SpaceFrame, as JiniMediators are not absolutely necessary to illustrate the most important concepts of the space-based architecture.

We have also developed an example application to demonstrate how to use the SpaceFrame prototype framework. The example application sends messages both with and without a direct addressing scheme, utilizing associative addressing. The application proved to be relatively easy to implement when having the SpaceFrame framework to build upon. When saying this, it is assumed that the developer is familiar with the modeling and principles of JavaFrame.

11.6 Further work

There are several issues in this thesis that could be subject to further work. To keep this section short, we only briefly present some areas of interest here.

- Security with Jini and JavaSpaces could be interesting to study in more depth. We see this as a key factor to the future success of Jini and JavaSpaces in open, dynamic environments. Also, a secure version of SpaceFrame could be made, utilizing the Davis project's [40] and the Yalta project's [43] efforts.

- Further work could be done on how JavaFrame-based architectures could benefit from the SOP paradigm [51] and Openwings' way of handling change and heterogeneity.

- Our space-based architecture is not tightly integrated with the current ActorFrame version, which is quite new. Rather, it is built upon JavaFrame, but both the actor and the role concept have been kept. This has been done because ActorFrame was not mature enough when we started our work. Further work should therefore try to align these two systems and updating the developed space-based conceptual architecture to fit new ideas from ActorFrame, including its Role request protocol, and ServiceFrame.

- Further work could also focus on improving the SpaceFrame prototype. This means implementing the Role&RoutingAgent properly and perhaps to develop a set of JiniMediators.

- Some of the concepts in the suggested space-based architecture could prove to be quite strong, for instance role trading and using transactions. A study of trading with JavaSpaces, with role trading as a case study could be interesting. One would then have to look at how a role's properties, for instance QoS capabilities, could be described by attributes.

12 Conclusion

In this thesis we have evaluated how Jini, JavaSpaces and frameworks building on these technologies could be used to distribute JavaFrame-based systems in dynamic environments. Until now, this problem has not been solved in a satisfactory way, but as we have seen in this thesis, using Jini and JavaSpaces might significantly ease this task.

We started out with an evaluation of the frameworks building on Jini and JavaSpaces with regards to JavaFrame. We then described four conceptual solutions combining Jini and JavaFrame, before describing our main work, a new space-based architecture for JavaFrame distribution. To show the usefulness of our ideas, we have also developed a prototype of the architecture called SpaceFrame and an example application that illustrates the use of SpaceFrame.

The work that has been done has shown that a new approach is needed when using the Jini and JavaSpaces technologies and frameworks building on these. One must adopt a view of software components as producers and consumers of services (Service-Oriented Programming).

Of the frameworks we have evaluated, Openwings stands out as the most mature component model that may be used to host JavaFrame-based components. Openwings has several concepts similar to JavaFrame concepts, and its container services can be utilized to ease the burden on developers. Our evaluation of the Rio architecture essentially shows that Rio does not have any additional features compared to Openwings. The space-based architecture SI is currently quite immature and poorly documented, but the concepts are strong, and SI can prove to be useful as middleware for JavaFrame-based systems when having matured.

Using Jini as middleware for JavaFrame-based systems opens up many possibilities. The software and protocols needed for communication with a service could be downloaded dynamically, which can be an elegant way to publish JavaFrame-based services. It is important to note that since the interaction still is of the peer-to-peer type, the application logic must be able to deal with failure, and find equivalent services. This is also the case when using Openwings or Rio.

Using JavaSpaces can provide asynchronous and uncoupled communication in time, space and destination, which are nice properties in dynamic environments where services and clients come and go all the time. Since JavaSpaces stores objects, both data and behavior, for instance role behavior, can be stored in the space.

We have found that using Jini and JavaSpaces together provides a comprehensive, yet simple toolkit for developing advanced distributed systems. Jini provides the infrastructure for finding JavaSpaces, while JavaSpaces is used a platform for interaction. Based on Jini and JavaSpaces, we have therefore developed a space-based

architecture and the SpaceFrame prototype. SpaceFrame is a framework for enabling JavaFrame-based systems to utilize JavaSpaces technology. The conceptual architecture and the SpaceFrame framework have proven that the two worlds of Jini/JavaSpaces and JavaFrame can integrate in an efficient and sensible way, using asynchronous communication end-to-end. The SpaceFrame ActorAgents' ability to play different roles on demand greatly enhances adaptivity. New concepts like the Role repository, role marketplaces and using transactions on roles and messages can also be valuable in dynamic environments.

A question which arises is whether our conceptual space-based architecture also has something to offer in environments where change occurs less frequently. Systems in such environments may also need a flexible way to adapt to change. Our impression is that the space-based architecture can be beneficial in almost any kind of environment, much due to the powerful concepts of role distribution and associative addressing. By using SpaceFrame, the developer's task of creating systems that adapt to change could be vastly simplified compared to using the traditional peer-to-peer style of interaction.

As commented in section 11.6, further work on how JavaFrame-based systems could benefit from the Service-Oriented Programming paradigm described in [51] could be done, as well as a study of securing Jini and JavaSpaces ([40] and [42]).

Abbreviations

ACID - Atomicity, Consistency, Isolation and Durability
ADL – Architecture Description Language
CORBA – Common Object Request Broker Architecture
CS – Composite State
DSM – Distributed Shared Memory
EJB – Enterprise Java Beans
FIFO – First-In First-Out
HTTPS - Hypertext Transfer Protocol over Secure Socket Layer
IIOP - Internet Inter-Orb Protocol
J2EE – Java 2 Enterprise Edition
J2ME - Java 2 Micro Edition
J2SE - Java 2 Standard Edition
JERI - Jini Extensible Remote Invocation
JMS – Java Messaging Service
JSB - Jini Service Bean
JVM – Java Virtual Machine
LAN – Local Area Network
LCD – Liquid Crystal Display
LUS – Lookup Service
MAC – Medium Access Control
MDK – Modeling Development Kit
OSA – Open Systems Architecture
PID – Protocol-Independent Design
PIP - Protocol-Independent Programming
PKI – Public Key Infrastructure
QoS – Quality of Service
RMI – Remote Method Invocation
RMI/IIOP – Remote Method Invocation over Internet Inter-Orb Protocol
RPC – Remote Procedure Call
SDL - Specification and Description Language
ServiceUI – Service User Interface
SI – Spanish Inquisition or Scalable Infrastructure
SMS – Short Message Service
SOA – Service-Oriented Architecture
SOAP – Simple Object Access Protocol
SOP – Service-Oriented Programming
SSL - Secure Socket Layer
TINI – Tini InterNet Interface
TLS – Transport Layer Security
UDDI - Universal Description, Discovery, and Integration
UI – User Interface
UML – Unified Modeling Language

UMTS – Universal Mobile Telecommunications System
WAP – Wireless Application Protocol
WSDL – Web Services Description Language
XML – Extensible Markup Language

References

1 Bræk, Rolf, Husa, Knut Eilif and Melby, Geir. *ServiceFrame Whitepaper, draft 22.04.2002,* Ericsson NorARC, 2002.

2 Haugen, Øystein and Møller-Pedersen, Birger. *JavaFrame: Framework for Java-enabled modelling*, ECSE2000, Ericsson NorARC, Stockholm, 2000.

3 Husa, Knut Eilif. *Serviceframe Software Architecture Document, Initial Version 29/01/02,* Ericsson NorARC, 2002.

4 Li, Sing. *Professional Jini*, Wrox Press Ltd., 2000.

5 Sun Microsystems, Inc. *Jini™ Specifications v1.2*, Available from: http://www.sun.com/jini/specs [Accessed April 8, 2002].

6 Microsoft Corporation. *Understanding Universal Plug and Play, WhitePaper*, 2000, Available from: http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc [Accessed April 8, 2002].

7 uddi.org. *UDDI Technical White Paper*, 2000, Available from: http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf [Accessed April 8, 2002].

8 Gong, Li. *Project JXTA: A Technology Overview,* 2001, Available from: http://www.jxta.org/project/www/docs/TechOverview.pdf [Accessed April 8, 2002].

9 Sun Microsystems, Inc. *Java Remote Method Invocation Specification,* Available from: http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html [Accessed April 8, 2002].

10 Couloris, George, Dollimore, Jean and Kindberg, Tim. *Distributed Systems: Concepts and Design (*3rd Edition), Addison-Wesley, 2001.

11 Winer, Dave. *XML-RPC specification,* Available from: http://www.xmlrpc.com/spec [Accessed April 8, 2002].

12 Sun Microsystems, Inc. *Rio Architecture Overview* http://www.sun.com/jini/whitepapers/rio_architecture_overview.pdf [Accessed April 8, 2002].

13 Bieber, Guy and Carpenter, Jeff. *Openwings – A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems (Rev 2.0),* Available from: http://www.openwings.org/download.html [Accessed April 8, 2002].

14 Sun Microsystems, Inc. *JavaSpaces™Service Specification 1.2*, 2001, http://www.sun.com/jini/specs/ [Accessed April 8, 2002].

15 Wyckoff, Peter. *TSpaces,* IBM Systems Journal, 1998 , Available from: http://www.research.ibm.com/journal/sj/373/wyckoff.html [Accessed April 8, 2002].

16 McClain, John W. F., Sun Microsystems, Inc., *An Introduction to JavaSpaces™ Technology,* Slides from JavaOne 2002 session 1403, Available from: http://servlet.java.sun.com/javaone [Accessed April 20, 2002].

17 GigaSpaces web page, Available at: http://www.gigaspaces.com [Accessed April 8, 2002].

18 W3C. *Simple Object Access Protocol (SOAP) 1.1,* 2000, Available from: http://www.w3.org/TR/SOAP/ [Accessed April 8, 2002].

19 IntaMission web page, IntaSpaces page, Available at: http://www.intamission.com/what_is_intaspaces.asp [Accessed April 8, 2002].

20 Java™ 2 Enterprise Edition web page, Available at: http://java.sun.com/j2ee/index.html [Accessed April 9, 2002].

21 Tspaces web page, Available at: http://www.almaden.ibm.com/cs/TSpaces/index.html [Accessed April 8, 2002].

22 Cisco Systems. *Spanish Inquisition,* Available from: http://www.sun.com/jini/whitepapers/si_whitepaper.pdf [Accessed April 8, 2002].

23 Haugen, Øystein. *JavaFrame 2.5 Modeling Guidelines*, Ericsson NorARC, 2001.

24 Tanenbaum, Andrew S., van Steen, Maarten. *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002. Jini section available online from: http://www.cs.vu.nl/~ast/books/ds1/samples.html [Accessed April 8, 2002].

25 Freeman, Eric, Hupfer, Susanne and Arnold, Ken. *JavaSpaces™ Principles, Patterns, and Practice.* Addison-Wesley, 1999.

26 Venners, Bill. *The ServiceUI API Specification, Version 1.1beta3,* Available from: http://www.artima.com/jini/serviceui/Spec.html [Accessed April 8, 2002].

27 Ducasse, Stéphane, Hofmann, Thomas and Nierstrasz, Oscar. *OpenSpaces: An Object-Oriented Framework For Reconfigurable Coordination Spaces*, Software Composition Group, Institut für Informatik (IAM), Universität Bern, 2000, Available from: http://www.iam.unibe.ch/~scg/Archive/Papers/Duca00dOpenSpaces.pdf [Accessed April 8, 2002].

28 MPI Forum. The Message Passing Interface (MPI) standard web page, Available at: http://www-unix.mcs.anl.gov/mpi/ [Accessed April 8, 2002].

29 Halter, Steven L. *JavaSpacesExample by Example*, Prentice Hall 2002 First chapter available from: http://vig.pearsoned.com/samplechapter/0130619167.pdf [Accessed18. april 2002]

30 The Rio project web page. Available for members of the Jini Community at: http://jini.org/ [Accessed April 8, 2002].

31 The Openwings web page. Available at: http://www.openwings.org [Accessed April 8, 2002].

32 The Bluetooth web page. Available at: http://www.bluetooth.com/ [Accessed April 20, 2002].

33 Microsoft's .Net web page. Available at: http://www.microsoft.com/net/defined/whatis.asp [Accessed April 8, 2002].

34 Sun's SunONE web page. Available at: http://www.sun.com/service/solutions/beyond/ [Accessed April 20, 2002].

35 ABLE Group, School of Computer Science. Carnegie Mellon University. *An Overview of Acme,* Available from: http://www.cs.cmu.edu/~acme [Accessed April 8, 2002].

36 The Java™ Messaging Service web page, Documentation available from: http://java.sun.com/products/jms/ [Accessed April 8, 2002].

37 The SI web page. Available for members of the Jini Community at: http://jini.org/ [Accessed April 8, 2002].

38 Mail correspondence with Jeff Carpenter, Software Engineer in the Openwings Team on April 22, 2002.

39 The Java™ 2 Standard Edition, release 1.4 web page, Documentation available from: http://java.sun.com/j2se/1.4/ [Accessed April 8, 2002].

40 The Davis project web page, Available for members of the Jini Community at: http://jini.org/ [Accessed April 8, 2002].

41 Gong, Li. *Java™ 2 Platform Security architecture, Version 1.1*, Sun Microsystems, Inc. 2000. Available from: http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-specTOC.fm.html [Accessed April 8, 2002].

42 Byrd, Gregory T., Gong, Fengmin, Sargor, Chandramouli and Smith, Timothy J. *Yalta: A Secure Collaborative Space for Dynamic Coalitions*, IEEE 2nd SMC Information Assurance Workshop, West Point, New York, 2001. Available from: http://www.anr.mcnc.org/projects/Yalta/yalta-ias2001.pdf [Accessed April 8, 2002].

43 The Yalta web page. Available at: http://www.anr.mcnc.org/projects/Yalta/Yalta.html [Accessed April 8, 2002].

44 Waldo, Jim. The End of Protocols, The Java Developer Connection™, Available from: http://developer.java.sun.com/developer/technicalArticles/jini/protocols.html [Accessed April 8, 2002].

45 Rowstron, Antony. *Using Agent Wills to Provide Fault-tolerance in Distributed Shared Memory Systems.* Microsoft Research Ltd. Cambridge, UK, 2000. Available from: http://research.microsoft.com/~antr/papers.htm [Accessed May 3, 2002].

46 The LighTS web page. Available at: http://lights.sourceforge.net/ [Accessed May 3. 2002]

47 The JavaBeans specification. Available from: http://java.sun.com/products/javabeans/docs/ [Accessed April 20. 2002]

48 The iButton web page. Available at: http://www.ibutton.com [Accessed April 20. 2002]

49 The Anhinga Project at Rochester Institute of Technology. Available at: http://www.cs.rit.edu/~anhinga/ [Accessed May 5, 2002].

50 The Java 2 Micro Edition web page. Available at: http://java.sun.com/j2me/ [Accessed April 20. 2002]

51 Bieber, Guy and Carpenter, Jeff, *Introduction to Service-Oriented Programming (Rev 2.1),* Available from: http://www.openwings.org/download.html [Accessed May 11, 2002].