Using JavaSpaces to create adaptive distributed systems

Fritjof Boger Engelhardtsen
Ph. D student,
Agder University College,
Faculty of Engineering and Science

Tommy Gagnes
Researcher,
Norwegian defence research establishment

Abstract

JavaSpaces' support for asynchronous and loosely coupled communication can be used to simplify creation of advanced services in dynamic network-centric environments. In such environments clients and services come and go all the time and system-components may dynamically be added and removed.

This paper describes how JavaSpaces can be utilized to create adaptive distributed systems. We present a space-based architecture where agents adapt to changing demands placed on the system by dynamically requesting their behavior from a JavaSpace.

We argue that using JavaSpaces technology is a simple way to create adaptive systems, due to JavaSpaces' ability to support messaging in dynamic environments as well as distribution of agent behavior.

Contact:

Fritjof Boger Engelhardtsen
Agder University College
Grooseveien 36, N-4876 Grimstad, Norway
Fritjof.B.Engelhardtsen@hia.no
http://fritjof.net

1. Introduction

Creating adaptive distributed systems in dynamic network-centric environments is a difficult task. Handling change such as changing IP addresses, changing numbers of clients and services and changing system configurations places large burdens on developers and system administrators.

In this paper, we will show how JavaSpaces and the space-based communication paradigm can be used to simplify the creation of adaptive distributed systems.

Compared to the traditional peer-to-peer interaction model, a space-based architecture has several beneficial features. A space-based agent system is said to be robust because one agent failing 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. Achieving adaptivity is simplified because agents may communicate without knowing each other's addresses. Agents may also communicate even if they are not executing at the same time. Since communication is anonymous and associative, a variable number of distributed agents can work together to solve a task.

Based on these properties we have developed a space-based architecture that can be used to simplify the creation of adaptive distributed systems. The space-based

architecture consists of agents that utilize JavaSpaces ability to distribute both messages and agent behavior.

2. Jini and JavaSpaces

Jini [1] is a specification that has been developed by Sun Microsystems. It builds on the Java programming language and specifies a self-healing, service-oriented distributed architecture for dynamic environments. Jini enables computers to find each other and use each other's (software) services on a network without prior information about each other or the protocol used [2], only about the service wanted. This is known as a service discovery mechanism, and with Jini, service discovery is based on mobile code, IP multicast and a lookup service.

To make Jini self-healing, leases are utilized. Nearly every registration or resource must be leased, that is, it must periodically 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 healthy resources continue to be published.

JavaSpaces [3] is a specification of a Java-technology 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 system developed by David Gelernter at the Yale University [4]. A tuple is a vector of typed values or fields that are used for matching other tuples in the shared memory.

A JavaSpace stores objects, which means that behavior as well as data can be stored. Distributed processes interact concurrently with the JavaSpace by using Java's Remote Method Invocation (RMI) facility. Objects are written, read or taken from a JavaSpace by using associative addressing. By associative addressing we mean that processes use template matching to determine which objects to read or consume from the space. Template matching is based on object type and attributes.

By using Jini's remote event model, processes can set up event registrations on a JavaSpace in order to receive notifications when certain objects are written to the space. This, in addition to the fact that a write-operation on the JavaSpace is non-blocking ensures fully asynchronous communication across a JavaSpace.

The use of space-based technologies can provide three forms of uncoupled communication, namely uncoupling in time, space and destination [5]. By this we mean the following:

Uncoupling in time: Tuples have a life-span that is independent of 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 features mentioned above can be used to enhance adaptivity and flexibility in dynamic environments where clients and services come and go all the time and system-components may dynamically be added and removed.

3. Architecture description

To fully exploit JavaSpaces' possibilities, we have designed our own space-based architecture (Figure 1). In chapter 4, several ideas are presented to show the benefits of using such an architecture.

In our space-based architecture, all communication is based on asynchronous associatively addressed messaging through a JavaSpace. Only agents are allowed to interact with a space. Three specialized agents are introduced, namely ActorAgents, ProtocolAgents and Role&RoutingAgents.

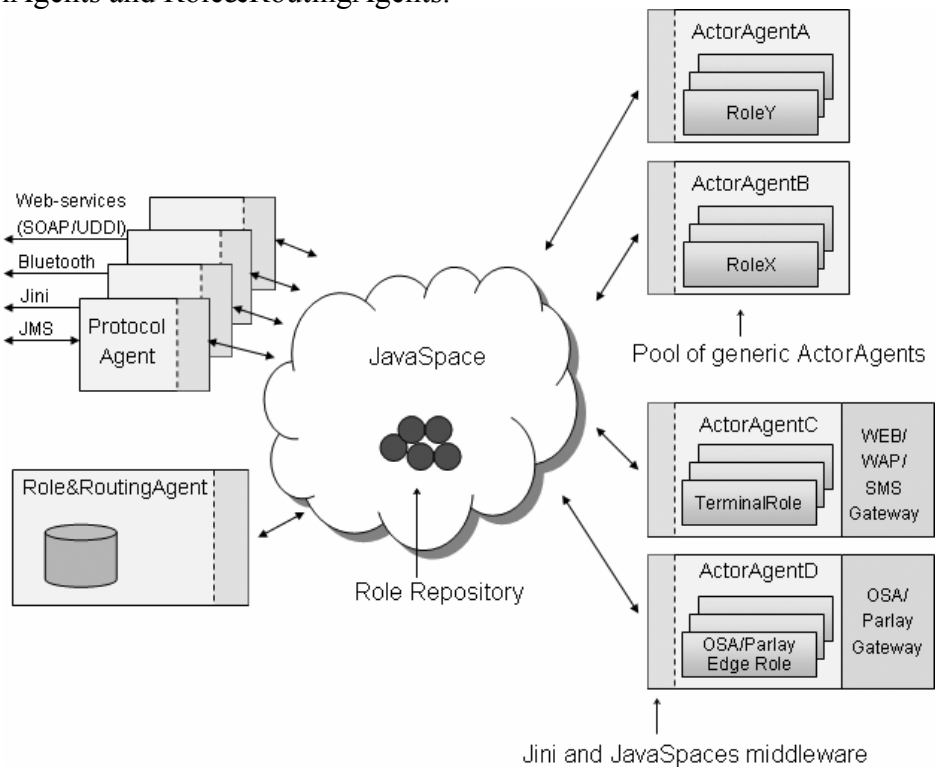

Figure 1. A space-based architecture

ActorAgents are agents that are capable of playing different *roles* within a domain. A role represents specific behavior. The ActorAgent "containers" provide Jini and JavaSpaces middleware to the role "components" hosted. Some ActorAgents represent specific entities in the real world, like SMS gateways, WAP servers or Service Capability Servers (typically OSA/Parlay gateways, location servers etc.) and therefore play specific roles. 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*. Using this type of agents will help systems adapt to changing demands placed on the system. This is achieved by ActorAgents requesting the proper roles from a *Role Repository*, which is a logical collection of stored objects that contain role behavior (executable code) for the roles played in the domain. Roles can time out after a given amount of time, freeing the ActorAgent hosting the role to play other roles that are requested in the domain. We call this concept a *Role timer*.

A *Role&RoutingAgent* knows which ActorAgents are capable of playing the different roles, and thereby the capabilities of the domain. If the Role&RoutingAgent cannot find the proper role in its own domain, it may forward messages to other domains (spaces). The Role&RoutingAgent is also responsible for the Role Repository mentioned above.

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.

Messages can either be addressed directly to the ActorAgent or they can be handled by an arbitrary ActorAgent qualified to process the message object. With *qualified* we mean that it plays 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 several ActorAgents try 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 specific ActorAgent by including an attribute that is known by both sender and receiver.

The presented architecture uses JavaSpaces both as a platform for communication and as an object store where role behavior can be stored. It may also be beneficial to use JavaSpaces to store and distribute other objects, like user profiles. Since processes can interact concurrently and asynchronously with a JavaSpace, different system parts may work at their own pace. In addition, using Jini's framework for remote events removes the need for processes to use multiple threads when communicating with multiple peers.

We believe that using Jini and JavaSpaces provides a comprehensive, yet simple toolkit for developing advanced distributed systems. Jini provides the infrastructure for finding JavaSpaces, while JavaSpaces is used as a platform for interaction between system components. Using Jini could provide an "always on" functionality where agents continually do service discoveries to find the available JavaSpaces in the context.

4. Advanced concepts

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

4.1 Transactions

By using the Jini distributed transaction model, one can ensure consistent execution of specific actions. This is based on the two-phase commit protocol, thereby making it possible to guarantee ACID properties. 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. Letting the role object and the message take part in a transaction may solve this problem. By doing this, the ActorAgent can either both read the role implementation *and* take the message (Figure

2), or do nothing. If it does nothing, the message will remain in the space, ready to be taken by any other agent able to play the necessary role.

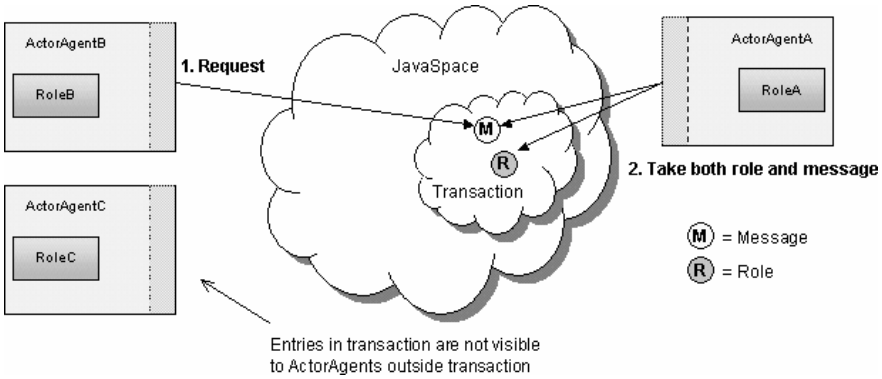

Figure 2. Using transactions in a JavaSpace

4.2 Role marketplaces

A common pattern used with JavaSpaces is the marketplace pattern. According to [6], "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.

The marketplace pattern could be used in several settings in our space-based architecture, but the most interesting use is probably trading roles and capabilities of ActorAgents. Roles could also be traded based on Quality of Service (QoS) parameters to ensure maximum utilization of the system resources. By using this pattern, ActorAgents can evaluate bids to find the role that is the closest match to their needs (Figure 3). Different roles could even be combined to provide the requested behavior.

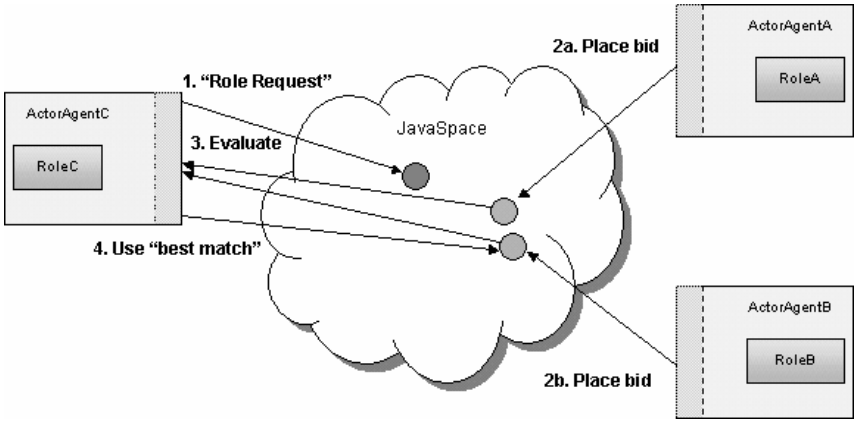

Figure 3. A role marketplace

4.3 Distributed data structures in a JavaSpace

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

In [6], 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 space-based 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 [6].

A channel in JavaSpaces terminology 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 (Figure 4). 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.

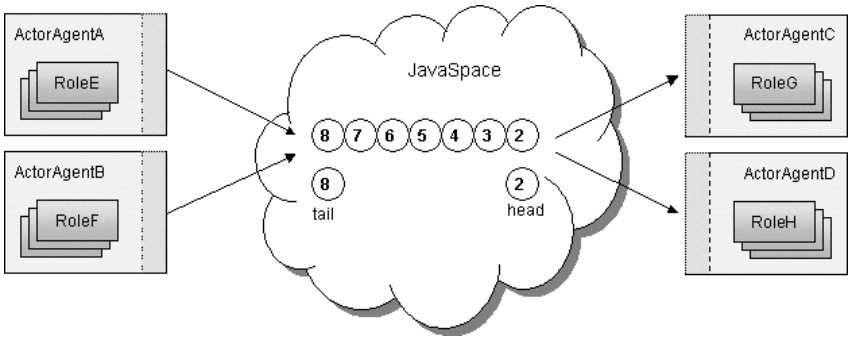

Figure 4. A channel

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

4.4 Using wills to ensure system consistency

ActorAgents are capable of establishing sessions across JavaSpaces by using addressed 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 can ensure that messages from a lost session will not stay in the space forever. Instead, they become garbage-collected when their lease is not renewed.

According to [7], «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 spacebased systems, but this alone might sometimes not be enough to reconstruct

application state and consistency in distributed data structures. In [7] 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 our space-based architecture, ActorAgents could use wills to ensure system consistency. A will in the space-based architecture could be represented by a role with a special will behavior, just like other roles. Should an ActorAgent fail, this role could 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 to detect failed agents is needed.

5. Scalability and performance

Realizing a space-based architecture with only a single JavaSpace introduces both a single point of failure and a bottleneck. However, this is not our intention. Using Jini ensures that agents can find another JavaSpace if the JavaSpace they are using fails. The Spanish Inquisition or Scalable Infrastructure (SI) [8] project by Cisco Systems is currently working on a highly scalable and robust architecture based on JavaSpaces. According to [8], SI 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 main idea behind SI is to organize a community of several JavaSpaces into a singe "virtual-space". Generally speaking, most work already done in the area of scalability and performance related to Tuple spaces and the Linda system is also applicable to architectures based on JavaSpaces.

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 space. 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. Doing this would combine the benefits of associatively addressed messaging with fast messaging within one JVM. A nice benefit of this approach is that it could be made transparent to the interacting ActorAgents whether they are located within the same JVM or not.

6. The SpaceFrame prototype framework

In this section we present an example where our space-based architecture has been used to distribute components from a service creation framework to dynamic environments.

To simplify the creation of complex services, Ericsson NorARC (Norwegian Applied Research Center) is developing the ServiceFrame [9] architecture, which is characterized as a «…service execution framework for advanced, hybrid and personalized services». ServiceFrame is a service execution framework that contains specific functionality for advanced telecommunication and Internet services. By specializing an already existing set of domain objects new services can be created rapidly. As shown in Figure 5, ServiceFrame is layered on top of the more general ActorFrame and JavaFrame frameworks.

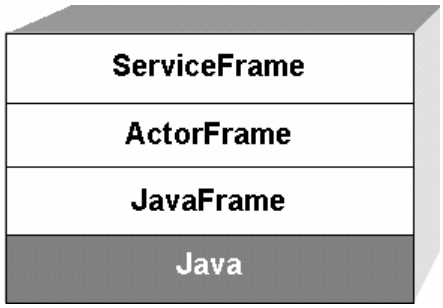

Figure 5. Framework dependencies

JavaFrame [10] 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. 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. To model the system behavior, the emerging UML 2.0 standard is used.

Complexity and composition is handled in ActorFrame by using an actor-abstraction. In ActorFrame, actors represent entities that can execute different behaviors, that is, play different roles.

We developed the SpaceFrame prototype framework to demonstrate how JavaFrame-based systems could benefit from JavaSpaces' ability to support loosely coupled, asynchronous communication and distribution of role behavior. Jini's framework for remote events is used to provide fully asynchronous communication across the JavaSpace.

We have implemented a set of SpaceMediators that use Jini in order to connect to JavaSpaces service. The SpaceMediators are associated with two sets of JavaFrame messages. One is used for communication through a JavaSpace, while the other is used for internal interaction between the ActorAgent and the SpaceMediators. Furthermore, we have implemented a basic ActorAgent that uses the SpaceMediators for interaction with a JavaSpace. Once the ActorAgent receives an initial message, it requests an appropriate role from the Role repository in order to be able to handle the received message. The behavior obtained from the Role repository is used to define the inner state machine (composite state) of the ActorAgent, thereby enabling it to play a specific role. Currently, 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.

Although SpaceFrame only demonstrates the core concepts of our space-based architecture, it can be seen as proof that such an architecture is possible to realize. End-to- end asynchronous communication is used, providing a smooth integration with JavaFrame concepts such as minimal thread usage. 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 and adapting to changing demands in the system.

7. Conclusions

Creating adaptive systems in dynamic environments where services and clients come and go all the time and system components may dynamically be added and removed is a complex task.

JavaSpaces has several features that can ease this task, including its ability to provide asynchronous and uncoupled communication in time, space and destination based on associative addressing. Since a JavaSpace stores objects, it is a simple means of distributing both messages and agent behavior.

Our space-based architecture utilizes these possibilities together with the actor-role abstraction to simplify the creation of adaptive systems. The architecture consists of three main types of agents that interact asynchronously through the space.

The ability of the space-based architecture's Generic ActorAgents 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.

The SpaceFrame prototype framework has been developed to provide JavaSpaces technology to systems based on Ericsson NorARC's JavaFrame technologies. The development of SpaceFrame has proven that our space-based architecture can simplify creation of systems that adapt to change. More work is however needed to determine how technologies for service-creation can benefit from using Jini and JavaSpaces as middleware.

8. References

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

[2] 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].

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

[4] *Generative Communication in Linda*, David Gelernter, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112 (January 1985).

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

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

[7] 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].

[8] Cisco Whitepaper, *Cisco Uses Jini Network Technology for Scalable Communication Framework*, Available from: http://www.sun.com/software/jini/whitepapers/ [Accessed August 1, 2002].

[9] Sanders, Richard Torbjørn, *Service-Centred Approach to Telecom Service Development*, EUNICE2002, Norwegian University of Science and Technology, 2002.

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