User and Session mobility in a Plug-and-Play architecture

by

Lars Erik Liljebäck

Norwegian University of Science and Technology
Department of Telematics
Trondheim, June 3rd, 2002

NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR INFORMATIKK, MATEMATIKK OG ELEKTROTEKNIKK

DIPLOMOPPGAVE - THESIS

Kandidatens navn:	Lars Erik Liljebäck
Fag:	Telematikk
Oppgavens tittel (norsk):	Bruker- og sesjonsmobilitet i en Plug-and-Play arkitektur
Oppgavens tittel (engelsk):	User and Session mobility in a Plug-and-Play architecture

Oppgavens tekst:
ITEMs Plug-and-Play (PaP) system architecture (Tapas) is realised by a platform that supports the basic PaP functionality. Extended functionality for User mobility and Session Mobility has been specified and shall be incorporated in the PaP basic platform functionality. The work comprises the design and implementation of PaP applications that both supports user mobility and session mobility.

Oppgaven gitt:	14. Januar 2002
Besvarelsen leveres innen:	7. Juni 2002
Besvarelsen levert:	
Utført ved:	NTNU
Veileder:	Stipendiat Mazen Malek

3. Juni, 2002

Finn Arve Aagesen

Abstract

Plug-and-Play (PaP) is a Java based software platform for the development of distributed, dynamic plugable applications. Dynamic plugable means that the physical architecture and the specification of the behaviour of a PaP application can be altered dynamically during the lifetime of the PaP application. The PaP model uses a theatre terminology, and two of the main concepts are actor and rolesession. An actor is a software entity playing a role, and two actors can communicate via a rolesession. A director actor supervises the actors in a PaP domain, administrates the plays (applications) plugged in and which roles each actor is playing. A PaP domain is the set of nodes or computers that are administrated by the same director.

In this thesis a partial framework for mobility in the PaP architecture was created by implementing support for user and session mobility in the PaP platform. User mobility means that users are not limited to use a certain terminal to access their services provided by their home domain. Session mobility means that they are not required to repeat several tasks in order to resume some unfinished work from previous sessions. A userprofile contains properties belonging to a user's profile, such as his username and password, while a sessiondescription contains information about a user's session, such as a list of the applications that he has started.

Functionality for supporting user and session mobility has been incorporated into the director actor. A director is also in charge of the userprofiles and the sessiondescriptions of the users of his domain. Two information bases, the UserProfileBase and the SessionBase, is used to store the sessiondescriptions and userprofiles of the users of a director's domain. Three agents provide a graphical user interface to the PaP system for the users. The loginagent is used to login a user to a domain, the visitoragent is provided for the visitors of a domain while the useragent is provided for users logged into their home domain. A useragent is able to suspend and later resume a user's session.

Two applications, the Chat and the FileTransfer application, have been created to demonstrate the capabilities of the mobility framework. They show that it provides user and session mobility within one domain and between different domains. A user is therefore able to logon to his home domain from another domain. The framework also lays out a foundation for supporting actor and terminal mobility, which is planned implemented in the near future.

Preface

This report is the diploma thesis written the spring of 2002 by Lars Erik Liljebäck. The objective was to create a partial framework for mobility in a Plug-and-Play (PaP) architecture by designing and implementing user and session mobility. PaP is a Java based software platform for the development of distributed, dynamic plugable applications. Dynamic plugable means that the physical architecture and the specification of the behavior for a PaP application can be altered dynamically during the lifetime of the PaP application.

Chapter one gives an introduction to the concepts used in the Plug-and-Play model, what functionality the PaP platform offers and how to setup and configure the network to use it. Chapter two gives an overview of the partial mobility framework to show how the result of this project fits within the overall mobility framework. Chapter three presents the requirements and the design of user and session mobility. It also explains the implementation of the mobility framework and shows the functionality that it offers. The mobility framework uses a set of XML-files and chapter four explains the purpose and the syntax of each of the XML-files used. Chapter five and six present the design and implementation of the Chat and the FileTransfer application, respecitively. They were created to demonstrate the capabilities of the mobility framework.

I would especially like to thank Mr. Mazen Malek Shiaa who I have cooperated with throughout this semester. Together we have had countless meetings discussing and resolving the issues around my work.

Trondheim June 3rd, 2002.

Lars Erik Liljebäck

Contents

Thesis i

Abstract iii

Preface v

1 Introduction 1

1.1 The Plug-and-Play model 2
1.1.1 The Plug-and-Play project 2
1.1.2 Definitions 2
1.1.3 The theatre metaphor 3
1.2 Overview of the Plug-and-Play system 5
1.2.1 The Plug-and-Play layered model 5
1.2.2 The director actor 6
1.2.3 Plug-and-Play system functionality 7
1.2.4 The Global Actor Identifier 9
1.3 How to use the Plug-and-Play system 11
1.3.1 Installation and configuration 11
1.3.2 Starting the Plug-and-Play system 12
1.4 Vocabulary of Plug-and-Play terms 14

2 The mobility framework 17

2.1 Mobility support in the PaP architecture 18
2.2 Focus on User and Session Mobility 20

3 The mobility architecture 23

3.1 Requirements to the mobility framework 24
3.1.1 Functional requirements 24
3.1.2 Non-functional requirements 24
3.2 Use cases 26
3.3 Screenshot 28
3.4 Class diagram 29
3.5 Sequence diagrams 31
3.5.1 Login user 31
3.5.2 Login user remotely 32
3.5.3 Logout user 33
3.5.4 Plugin actor 34

3.5.5 Plugout actor 35
3.5.6 Suspend session 36
3.5.7 Resume session 37
3.5.8 Update session 38

4 The XML files **39**
4.1 What is their use 40
4.1.1 Why do we need to store information 40
4.1.2 What is XML and why was it chosen 40
4.2 The file RoleList.xml 41
4.3 The file SessionDescriptions.xml 42
4.4 The file UserProfiles.xml 44

5 The Chat application **45**
5.1 Requirements to the Chat application 46
5.1.1 Functional requirements 46
5.1.2 Non-functional requirements 47
5.2 Use cases 48
5.3 Screenshot 49
5.4 Class diagram 50
5.5 Sequence diagrams 51
5.5.1 Connect 51
5.5.2 Disconnect 52
5.5.3 Get userlist 53
5.5.4 Send and receive message 54
5.5.5 Suspend session 55
5.5.6 Resume session 56

6 The FileTransfer application **57**
6.1 Requirements to the FileTransfer application 58
6.1.1 Functional requirements 58
6.1.2 Non-functional requirements 59
6.2 Use cases 60
6.3 Screenshot 61
6.4 Class diagram 62
6.5 Sequence diagrams 64
6.5.1 Send filetransfer request 64
6.5.2 Send filepart 65
6.5.3 Cancel transfer 66
6.5.4 Suspend transfer session 67
6.5.5 Resume transfer session 68

7 The results **69**
7.1 Problems encountered 70
7.1.1 Plugging out actors 70
7.1.2 Discrimination of certain types of requests 70
7.1.3 Deadlock of threads 71

7.1.4 Introducing new types of requests 71
7.1.5 Start-up of the PaP system 72
7.2 Testing 73
7.3 Future improvements 74

Conclusion **75**

Bibliografy **77**

Appendix A - Testcases **79**
Testing user mobility 79
Testing session mobility 80

List of Figures

1.1 The PaP data model. 3
1.2 Illustration of the theatre terminology. 4
1.3 Example of a PaP system. 5
1.4 The director actor and the information bases. 6
1.5 PaP system functionality. 7
1.6 More PaP system functionality. 8
1.7 The Global Actor Identifier adressing schema. 9
1.8 Screenshot of a PNES' baseframe window. 12

2.1 Object model for Mobility support in the PaP architecture. 18
2.2 General Engineering model for mobility support in the PaP architecture. . 19
2.3 Session mobility: A user is interacting with the system through useragent and its session is stored in the director's SessionBase. 21
2.4 User mobility: User A is accessing his home domain from Domain2. 21

3.1 Use cases for the design of user and session mobility. 26
3.2 Screenshot of the agents used in the mobility framework. 28
3.3 Class diagram for the mobility framework. 30
3.4 Sequence diagram for logging in a user to PaP domain. 31
3.5 Sequence diagram for logging in a user from another PaP domain. 32
3.6 Sequence diagram for logging out a user from a PaP domain. 33
3.7 Sequence diagram for plugging in an actor. 34
3.8 Sequence diagram for plugging out an actor. 35
3.9 Sequence diagram for suspending a user's session. 36
3.10 Sequence diagram for resuming a user's session. 37
3.11 Sequence diagram for updating a user's session. 38

5.1 Use cases for the design of the Chat application. 48
5.2 Screenshot of the Chat application. 49
5.3 Class diagram for the Chat application. 50
5.4 Sequence diagram for connecting a chatclient to a chatserver. 51
5.5 Sequence diagram for disconnecting a chatclient from a chatserver. 52
5.6 Sequence diagram for obtaining a list of all connected chatclients. 53
5.7 Sequence diagram for sending and receiving messages. 54
5.8 Sequence diagram for suspending a chatclient's session. 55
5.9 Sequence diagram for resuming a chatclient's session. 56

6.1 Use cases for the design of the FileTransfer application. 60

6.2 Screenshot of the FileTransfer application. 61
6.3 Class diagram for the FileTransfer application. 63
6.4 Sequence diagram for sending a filetransfer request. 64
6.5 Sequence diagram for transfering a file. 65
6.6 Sequence diagram for cancelling a filetransfer. 66
6.7 Sequence diagram for suspending a ftclient's session. 67
6.8 Sequence diagram for resuming a ftclient's session. 68

Chapter 1

Introduction

Contents

1.1 The Plug-and-Play model 2
1.1.1 The Plug-and-Play project 2
1.1.2 Definitions 2
1.1.3 The theatre metaphor 3
1.2 Overview of the Plug-and-Play system 5
1.2.1 The Plug-and-Play layered model 5
1.2.2 The director actor 6
1.2.3 Plug-and-Play system functionality 7
1.2.4 The Global Actor Identifier 9
1.3 How to use the Plug-and-Play system 11
1.3.1 Installation and configuration 11
1.3.2 Starting the Plug-and-Play system 12
1.4 Vocabulary of Plug-and-Play terms 14

1.1 The Plug-and-Play model

1.1.1 The Plug-and-Play project

In 1998 ITEM (Department of Telematics) at NTNU (Norwegian University of Science and Technology) started a project called "Plug-and-play for Network and Teleservice Components". The project is supported by The Norwegian Research Council. The vision of the project is to develop a flexible architecture that can provide Plug-and Play (PaP) functionality for different types of telecommunication equipment, as well as telecommunication networks and services.

Technical telecom solutions are steadily becoming more heterogenous, complex and diverse. Meanwhile qualified personnel is the critical factor for both development, installation and deployment, as well as operation, maintenance and evolution of telecom and teleservice software. The challenging question of how to handle this situation was the motivation behind the project. Its goal is that this architecture will both simplify and significantly speed up the installation and management, as well as the evolution and maintenance of telecommunication equipment and services.

1.1.2 Definitions

According to [1]:

> "Plug-and-play for network and teleservice components means that the hardware and software 'parts', as well as complete network elements that constitute a communication system have the ability to configure themselves when installed into a network and then to provide services according to their own capabilities, the service repertoire and the operating policies of the system."

A more simple explanation of the concept is given by [2]:

> "PaP simply means that you plug-in and then the system works."

Further, [3] defines a PaP component as:

> "some real-world active software or software/hardware module."

He distinguishes between static PaP and dynamic PaP, and defines static PaP as situations where:

> "components configure themselves at installation and provide services according to its predefined functionality"

and dynamic PaP in situations where:

> "components have a set of basic capabilities. Their functionality is decided during the plug-in procedure and can dynamically be changed during the lifetime of the component."

The Plug-and-Play project at ITEM is aiming to provide architecture for a dynamic PaP system. According to [2] the project members think that the use of dynamic PaP will increase the complexity and therefore also the cost of developing networks and services. However, since the highly dynamic and heterogeneous networks are evolving so rapidly, they foresee that the use of dynamic PaP is needed to keep the networks manageable. They expect that the benefits in deployment, installation, operation, management, maintenance and evolution outweigh the extra cost that PaP introduces.

1.1.3 The theatre metaphor

The specification of the needed PaP support functionality is defined through a theatre terminology. In this terminology there exist different concepts like plays, roles, manuscripts and actors. A more thorough description of the theatre metaphor and how it is used in the PaP architecture is given in [2, page 327-330]. In the article the theatre terminology is described this way:

> ”Plays define the functionality of the system. PaP components are realised by actors playing roles defined by manuscripts. An actor's capabilities define his possibilities for playing various roles.”

Figure 1.1 shows the data model and the relation between the different concepts used in the PaP architecture. A manuscript defines how a role is to be played. It defines the total behavior of an object, which PaP entities to cooperate with and how to reach them. When an actor is assigned a role, an instance of the role's manuscript is created and sent over the network to the actor. The functionality that the manuscript defines is then plugged into the actor who starts playing the role.

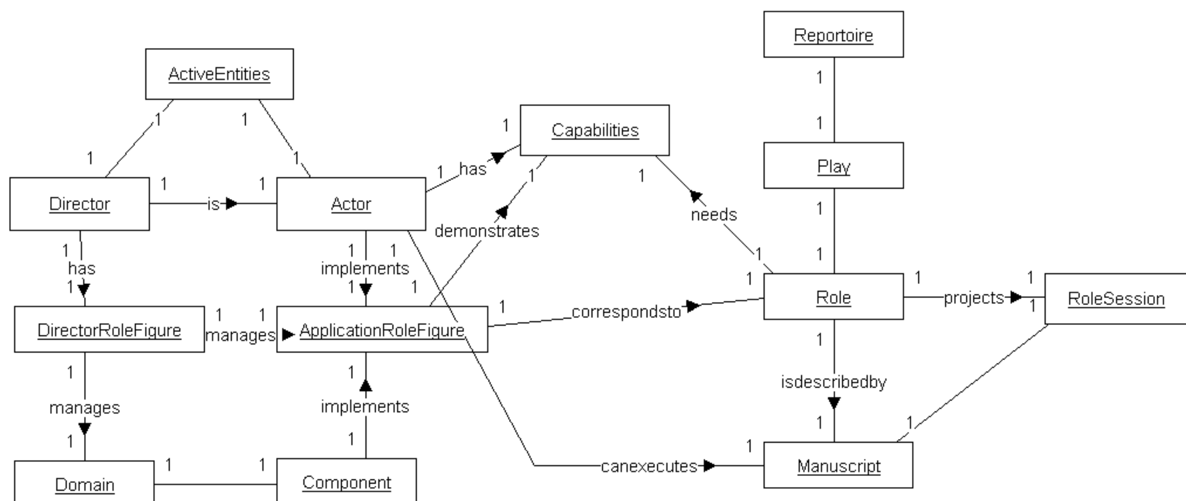

Figure 1.1: The PaP data model.

To distinguish between an actor and a rolefigure (or application-rolefigure) can be difficult. In the real world an actor is a person. When the actor plays a role in a play he becomes a rolefigure. However, when we transform these concepts to the digital world, there is no such thing as a plain actor. Since an actor always plays a role every actor is in fact a rolefigure. Figure 1.2 [3] illustrates the theatre terminology.

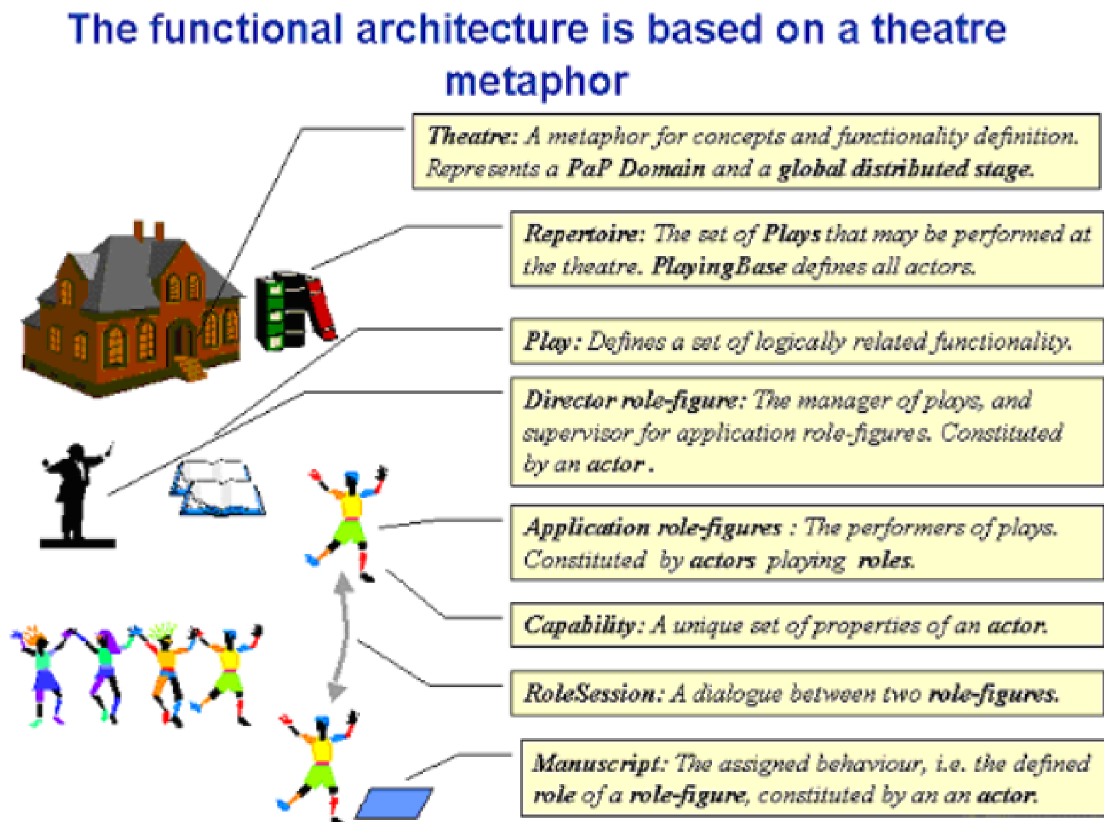

Figure 1.2: Illustration of the theatre terminology.

1.2 Overview of the Plug-and-Play system

1.2.1 The Plug-and-Play layered model

The functionality that the PaP system provides is structured in several layers. The bottom layer is the PaP Communication Infrastructure (PCI). It provides some basic communication capabilities. The next layer is the PaP Node Execution Support (PNES). A node can for example be a computer. There is only one PNES instance in each node. The PNES layer enables a node running PaP software. It also enables PaP entities on different nodes to communicate with each other. The third layer is the PaP Actor Support (PAS). A PAS makes it possible to create actors and assign behavior to them. It also lets the actors communicate with their environment. Figure 1.3 [3] shows an example of a PaP system. The system consists of four nodes where three of them have an instance of PNES enabling them to run PaP software. A thorough description of the layered model and the two top layers is given in [4].

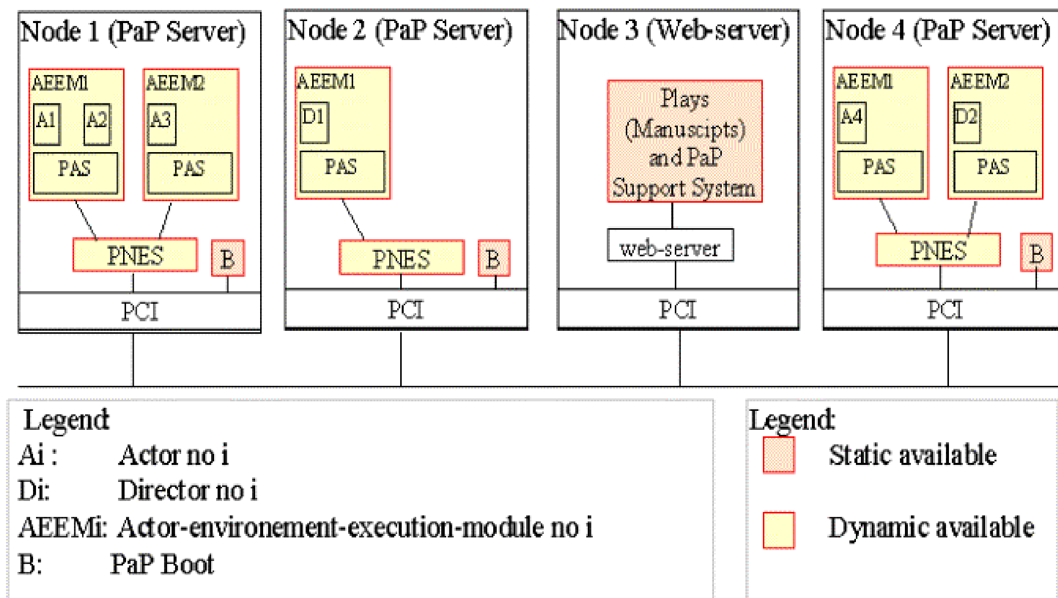

Figure 1.3: Example of a PaP system.

1.2.2 The director actor

The director is an actor with extra responsibilities. Figure 1.4 shows how the director communicates with other actors. It controls and administrates the domain by maintaining three information bases. The PlayingBase is a list of the different active actors in the domain (the actors that are playing). The RepertoireBase is an overview of the available plays. The ManuscriptBase contains the manuscripts for the different roles used in the plays in the RepertoireBase.

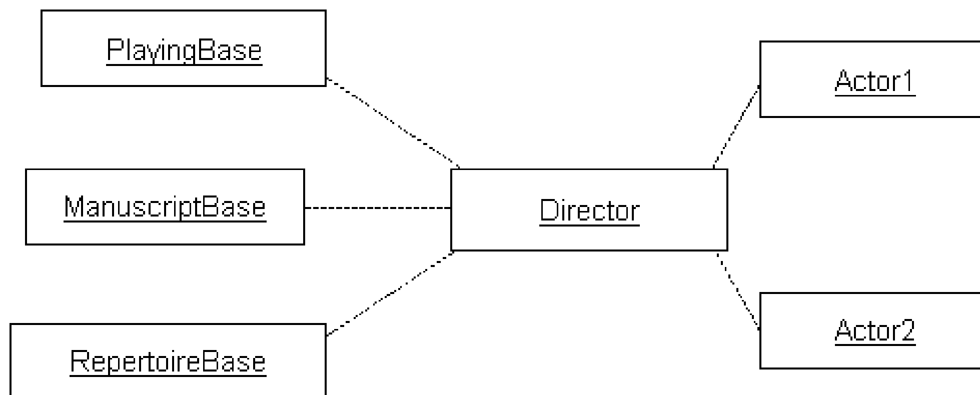

Figure 1.4: The director actor and the information bases.

When an actor is to play a role, the director fetches the specified manuscript from the ManuscriptBase and sends it to the actor over the network. The actor then starts playing the role according to the manuscript. When an actor wants to use part of the PaP support functionality, it sends a request to the director who either performs a set of operations or forwards the request to the appropriate PaP entity. This way the director controls most of the operations performed in his domain.

Every request a PaP entity receives from another entity it answers by returning a Request-Result. It has two types, OK and FAIL, indicating whether the entity was able to serve the request or not. A RequestResult contains a property called resultCause. This can optionally be used, by the entity serving the request, to notify the sender of the cause of the returned result.

1.2.3 Plug-and-Play system functionality

Figure 1.5 [3] shows the system functionality offered by the PaP system. For a more detailed description of the functionality see [2, page 330-331]. A play can be plugged in from the RepertoireBase and made available via the director. It can also be plugged out or changed. When an actor is plugged in a generic actor is created and initialized. Then it receives its functionality from the director in the shape of a manuscript object. The actor's behaviour can be changed so that it receives a new manuscript object or the behavior can simply be plugged out. The whole actor can also be plugged out.

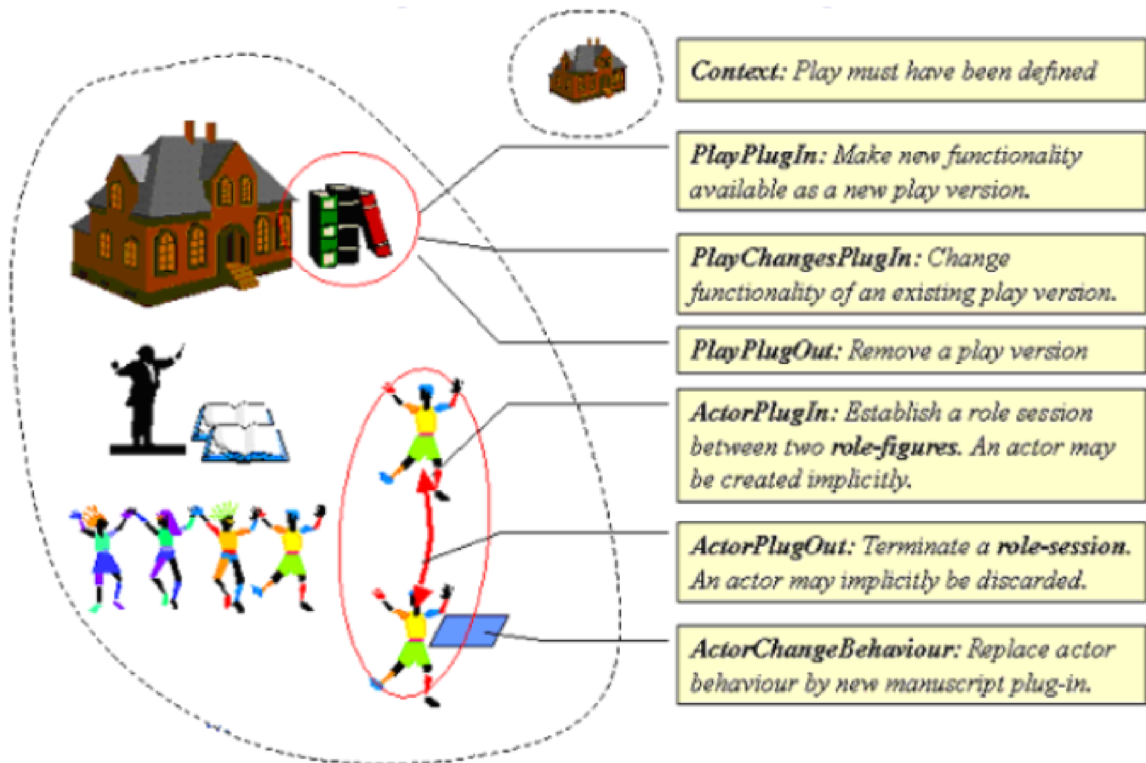

Figure 1.5: PaP system functionality.

Figure 1.6 [3] shows some additional functionality offered by the PaP system. PaP entities can communicate with each other by sending RoleSessionAction requests. It is a special type of request sent via a rolesession and contains a message referred to as an Application-Message.

An actor can also subscribe to one or more types of events occurring in the PaP system with the director. If one or more of those types of events occurs the actor will receive detailed reports about them. One of the example applications utilizing the PaP system functionality is the Watcher application. It is a monitoring application for the PaP system. Watcher simply subscribes to every types of events defined, receives reports from the director and shows them to the user.

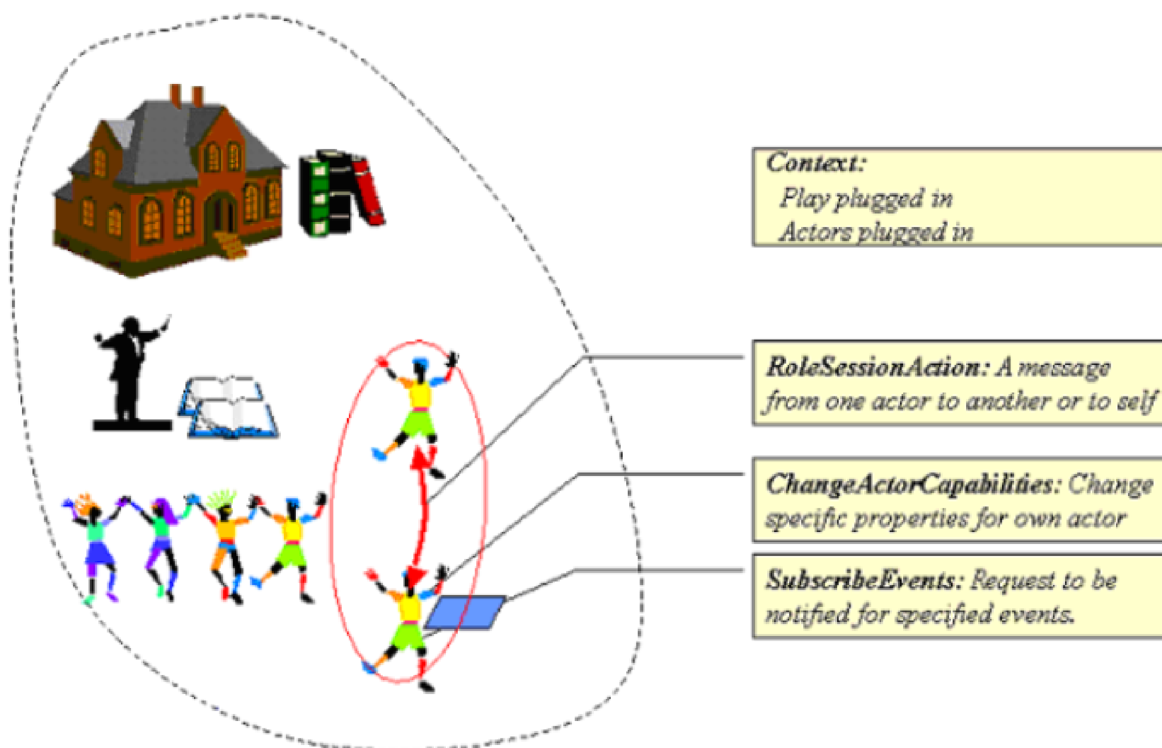

Figure 1.6: More PaP system functionality.

1.2.4 The Global Actor Identifier

The PaP architecture identifies every addressable entity through Global Actor Identifiers (GAI). There are four addressable types defined; PNES, PAS, Actor and RS (RoleSession). A GAI consists of several parts specific to the addressable type. A PNES instance is uniquely identified by the PNES identifier value. A rolesession, on the other hand, is identified by a PNES identifier, a PAS identifier, a director actor identifier and a unique number within the specified director actor. By using this address scheme every entity instance can be uniquely identified globally. Figure 1.7 [3] shows what parts are used to form a GAI for each of the PaP entity types.

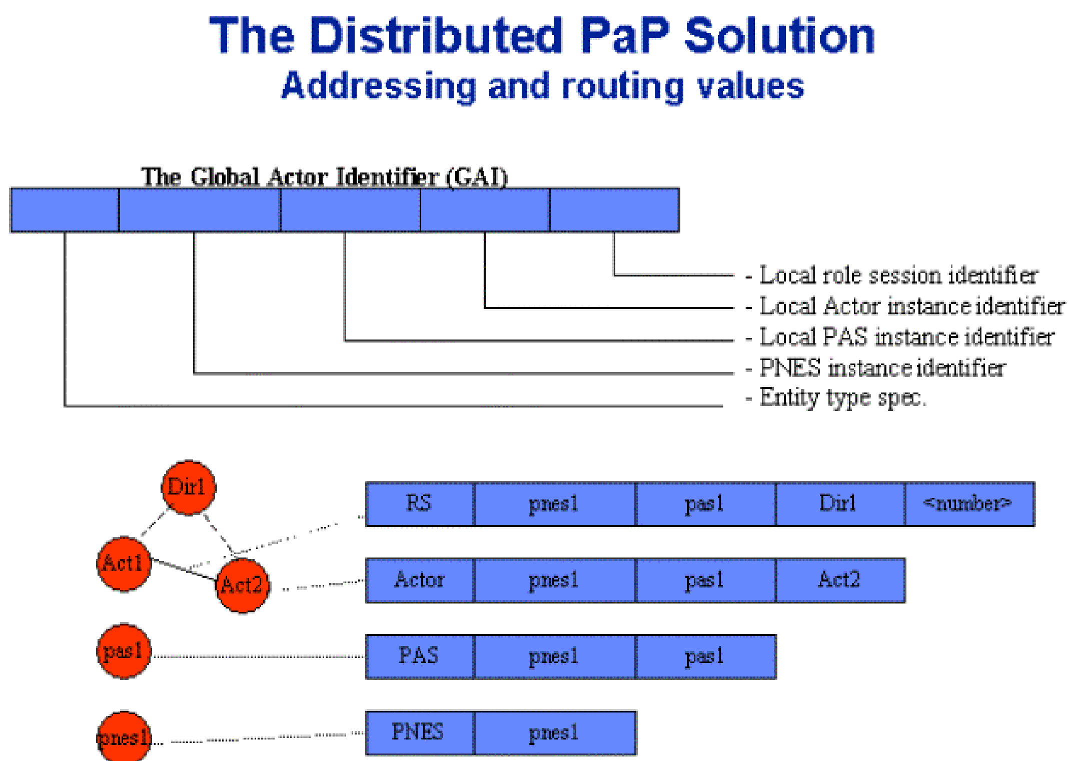

Figure 1.7: The Global Actor Identifier adressing schema.

Examples of a GAI for each of the addressable types are given underneath:

- PNES://129.241.200.175
- PAS://129.241.200.175/pas1
- Actor://129.241.200.175/pas1/W1
- RS://129.241.200.175/pas1/PaP.Director1/2

1.3 How to use the Plug-and-Play system

This section is aimed as a quick guide on how to install, configure and start the PaP system. The first part describes the steps necessary to install and configure your network, while the second section describes how to plug-in plays and actors. Throughout the thesis the operating system is assumed to be Microsoft Windows.

1.3.1 Installation and configuration

Different types of PaP software can be downloaded from [5]. This includes the PaP boot software and several example applications. In order to setup a network to run PaP software one must go through the steps listed below.

1. Download the PaP software to be used.
2. Make the plays to be used and a configuration file available from a webserver.
3. Install the PaP boot software and configure the pap.cnf file on every node that shall be able to run PaP software.

The PaP plattform is implemented in the Java programming language. The source code is compiled into class-files. The class-files must be put on a webserver for the PaP system to download when needed. Each play has its own package where the playname is the super-package and the version of the play is the subpackage. The Chat application (see chapter 5) is a play. The class-files for the application is put in a directory called v1_1. This is the subpackage and specifies the version of the play, which is 1.1. This directory is located in another directory called Chat, which is the superpackage and specifies the name of the play.

The PaP system uses a technology called Java RMI (Remote Method Invocation). Part of this technology is a so called RMISecurityManager handling security issues like access rights on a computer. One must specify to him the permissions to be granted to the PaP system on a node. This is done by making a file called policy available from a webserver. An example of a policy file is listed below.

```
grant {
  permission java.security.AllPermission;
};
```

The file PaPBoot.zip contains the PaP boot software. Unzip it in a directory on your computer. The zip-file creates a directory called StartClient which contains a file called pap.cnf and a directory called PaP. The PaP directory contains the boot software. The pap.cnf file is a configuration file read by the PaP boot software when it is started. The PNES on the node is configured according to the information in the file. An examle of such a file is listed below.

```
codebase = http://www.stud.ntnu.no/~liljebac/pap/
policy = http://www.stud.ntnu.no/~liljebac/pap/policy
homeinterface = Actor://129.241.200.227/pas1/PaP.Director1
debugserver = localhost
nodeprofile = profDefault
```

- **Codebase** - specifies the directory on a webserver that contains the class-files for the applications and the support functionality. According to the example listed above the class-files for the Chat application would be located in the directory **http://-www.stud.ntnu.no/~liljebac/pap/Chat/v1_1/**.
- **Policy** - specifies where the policy file is located.
- **HomeInterface** - specifies the GAI of the director actor for the PNES on the node.
- **DebugServer** - specifies where an instance of a DebugServer is located. A DebugServer provides functionality for debugging the PaP system. It can be started by running the command *java PaP.startPaP PaP.DebugServer* on the commandline.
- **NodeProfile** - is used by applications for additional settings.

1.3.2 Starting the Plug-and-Play system

Start a MS-DOS terminal and go to the directory where the PaP boot software is installed. Type the command *java PaP.startPaP PaP.PNES* and press Enter. A PNES instance will be started and the PNES' baseframe window will popup. A screenshot of a PNES' baseframe window is shown in figure 1.8. A baseframe window is an interactive and primitive user interface to the PaP entity that the window belongs to. Every active entity has one. Debug information from the entity is shown in the window and commands can be sent to it by typing them in the textfield and pressing Enter. A nice feature is that all commands written in the baseframe window are stored. By pressing the up or down arrow you can cycle through the current set of commands.

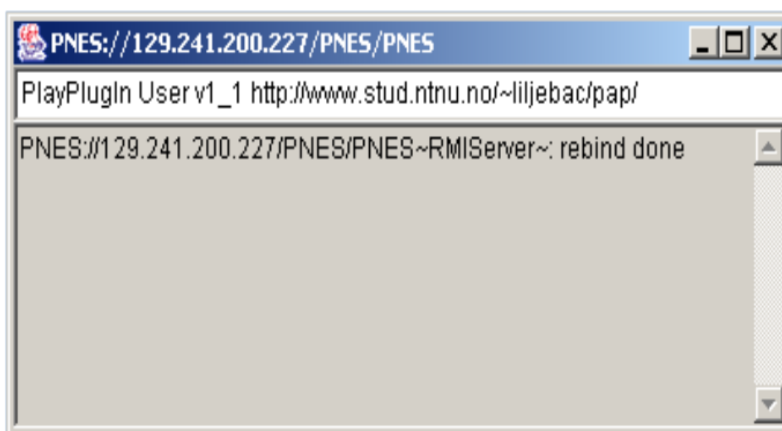

Figure 1.8: Screenshot of a PNES' baseframe window.

The first thing we need to do is to plugin a play. No actor can be plugged in unless the play of the role assigned to the actor is plugged in first. In other words, no Chat-Client can be plugged in unless the play Chat is plugged in. To plugin a play, type the command *PlayPlugIn <play-name> <play-version> <codebase>* in the PNES' baseframe window and press Enter. The PNES wants to send the specified request to his director or homeinterface. Since only the PNES is running a number of operations is performed. The director is assumed to run on the same node as the PNES. If it is running on another node the request is sent to the PNES on the director's node and the same operations mentioned will be performed there.

The GAI of the director is the one written in the pap.cnf file as the value of the Home-Interface property. Based on this GAI the PNES checks if the director's PAS is running. If it is not running, a new PAS is created and started, before the request is forwarded to it. The PAS checks if the receiving actor, in this case the director, is plugged in. If it is not a new director actor is created and plugged in by sending it an ActorPlugIn request. When the director receives the request it first initializes itself before the specified play is finally plugged in. Note that the first PlayPlugIn request takes a while to be served, as mentioned later in section 7.1.5.

Now that a play is plugged in we can plugin an actor with one of the roles of the play. To plugin an actor type the command *ActorPlugIn <actor-gai> <role-name>* in the PNES' baseframe window and press Enter. The request is sent to the director who pluggs in an actor on the location specified by actor-gai. A list of some of the commands defined can be found in [6, page 10-13]. Note that not all commands or requests can be served by every PaP entity.

1.4 Vocabulary of Plug-and-Play terms

A vocabulary of the most common PaP terms is listed below.

- Actor - A software entity. An actor can be plugged into any node in a network with a PNES instance running. When an actor is created it is assigned a role which can at anytime be changed to another role.
- ApplicationMessage - Message sent in a RoleSessionAction request via a rolesession between two PaP entities.
- BaseframeWindow - An interactive debugging window that every active PaP entity has. It shows status and debugging information about the entity, and through the window one can send requests to it.
- Capability - A unique set of properties of an actor.
- Director - Specialized actor that administrates a PaP domain and three bases with different sets of information (ManuscriptBase, PlayingBase and RepertoireBase).
- GAI - Global Actor Identifier is the adressing scheme used in the PaP architecture. Used to identify every addressable entity. Four types defined: PNES, PAS, Actor and RS (RoleSession).
- HomeInterface - Every PNES, PAS and actor has a reference to a HomeInterface which is the GAI of the Director of the entity's PaP domain.
- Manuscript - Implementation of a role that defines the role's behaviour. An actor is assigned its role by plugging in an instance of the role's manuscript.
- ManuscriptBase - Contains the manuscripts for the different roles used in the plays in the RepertoireBase.
- Node - Piece of hardware running an instance of PNES. A node is usually a computer.
- PaP domain - The complete set of nodes and PAS instances that has a common HomeInterface or Director.
- PAS - PaP Actor Support. Layer in the PaP layered model that enables the creation of actors and enables roles to be assigned to actors.
- Play - Defines the functionality of a system or an application. This functionality is divided into a set of roles that together constitute the play.
- PlayingBase - List of the different active actors in a PaP domain, i.e. the actors that have been plugged in.
- PNES - PaP Node Excecution Support. Layer in the PaP layered model that enables a node running PaP software. It also enables PaP entities on different nodes to communicate with each other.
- Repertoire - Plays that can be plugged in.

- RepertoireBase - List of the available plays, i.e. the plays that can be plugged in
- RequestResult - As a response to every request sent in the PaP system a RequestResult is returned. It indicates whether the request could be served or not. Optionally the cause of the result can be added.
- Role - The name of an actor's behaviour. Every actor is assigned a role.
- Rolefigure - A rolefigure performs the whole or part of a play. Constituted by an actor playing a role.
- RoleSession - Logical connection between two PaP entities that enables them to communicate with each other. Every rolesession has an initiator, i.e. the PaP entity that initiated the creation of the rolesession, and a cooperator. When an actor is plugged in an initial rolesession is always created where the actor is the cooperator and the entity requesting the actor to be plugged in is the initiator. An actor can have a number of rolesessions depending on the role it is playing.
- RoleSessionAction - Type of request used to send a message between two PaP entities.
- Theatre - A metaphor for concepts and functionality definition. Represents a PaP domain and a global distributed stage.

Chapter 2

The mobility framework

Contents

2.1 Mobility support in the PaP architecture 18
2.2 Focus on User and Session Mobility 20

2.1 Mobility support in the PaP architecture

Generally speaking, mobility support in the PaP architecture is intended to expand to four generic categories: User, Session, Actor and Terminal mobility. User and Session mobility aims at providing the very basic personal mobility for users of any PaP system or application. Users are no longer limited to use certain terminal to access their services provided by their home domain, neither are they required to repeat several tasks in order to resume some unfinished work from previous sessions. Actor mobility is the support provided for applications to carry out their tasks in a flexible and optimum manner. This could be altered as software mobility as actors are analogous to software functionality in the PaP architecture. Terminal mobility, on the other hand assures the continuity of service access while on the move or at location change. This is a complex and rather circumstantial issue that heavily depends on network configuration and terminal capabilities. More information about the four generic categories of mobility described can be found in [7] and [8].

To get a wide-ranging view of mobility management in the PaP architecture an object model and an engineering model should be developed to comprise all needed functionality and support. The figures 2.1 and 2.2 depict how mobility is supported in the PaP architecture. In figure 2.1 all objects needed for various mobility kinds are illustrated and related to each other.

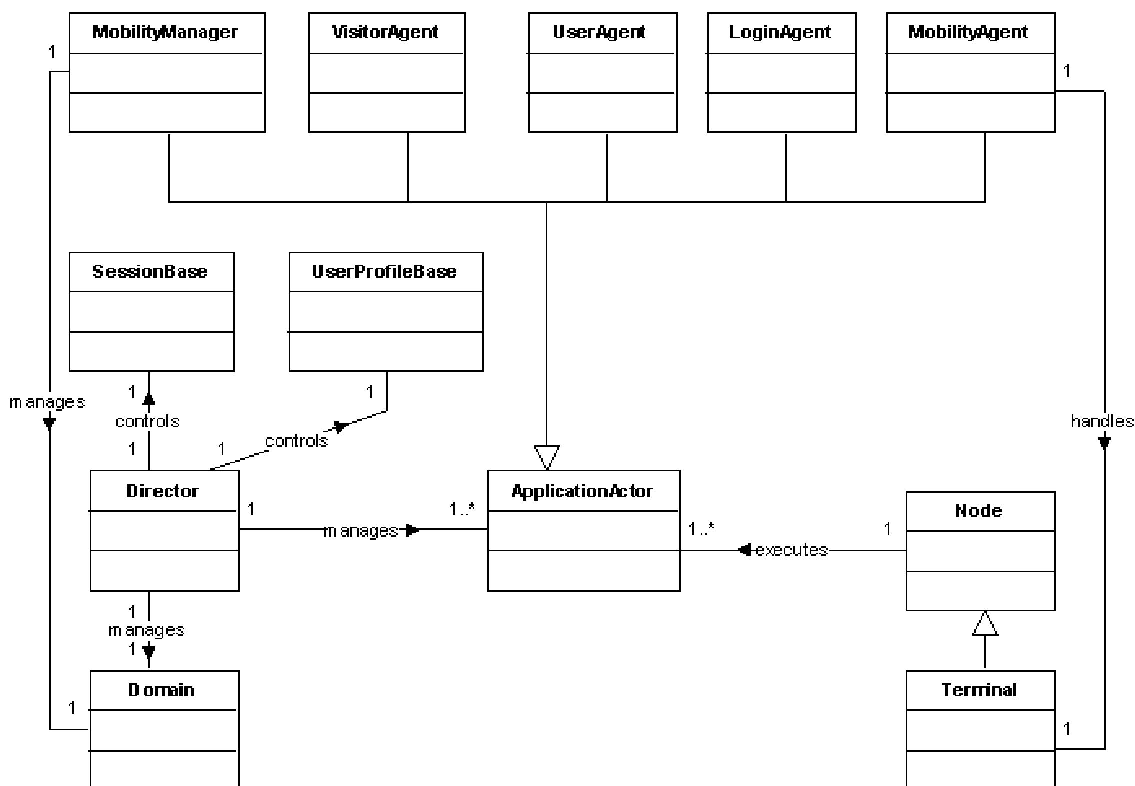

Figure 2.1: Object model for Mobility support in the PaP architecture.

Figure 2.2 gives an engineering model for mobility management. Note that the playing, session and userprofile bases are all controlled by the director. Additionally, Mobility Manager and the director could also be co-located in the same node. The devices depicted are characterized by different sets of capabilities. That is why certain application components run at networks nodes instead of user's device. The Mobility Manager is responsible for managing the terminal mobility of wireless devices. In such a configuration any network node could be mobile and being managed by it. For purpose of multi-domain environments the domain's director, as there is only one director per domain, could contact other directors inquiring about visiting user's ID and profile.

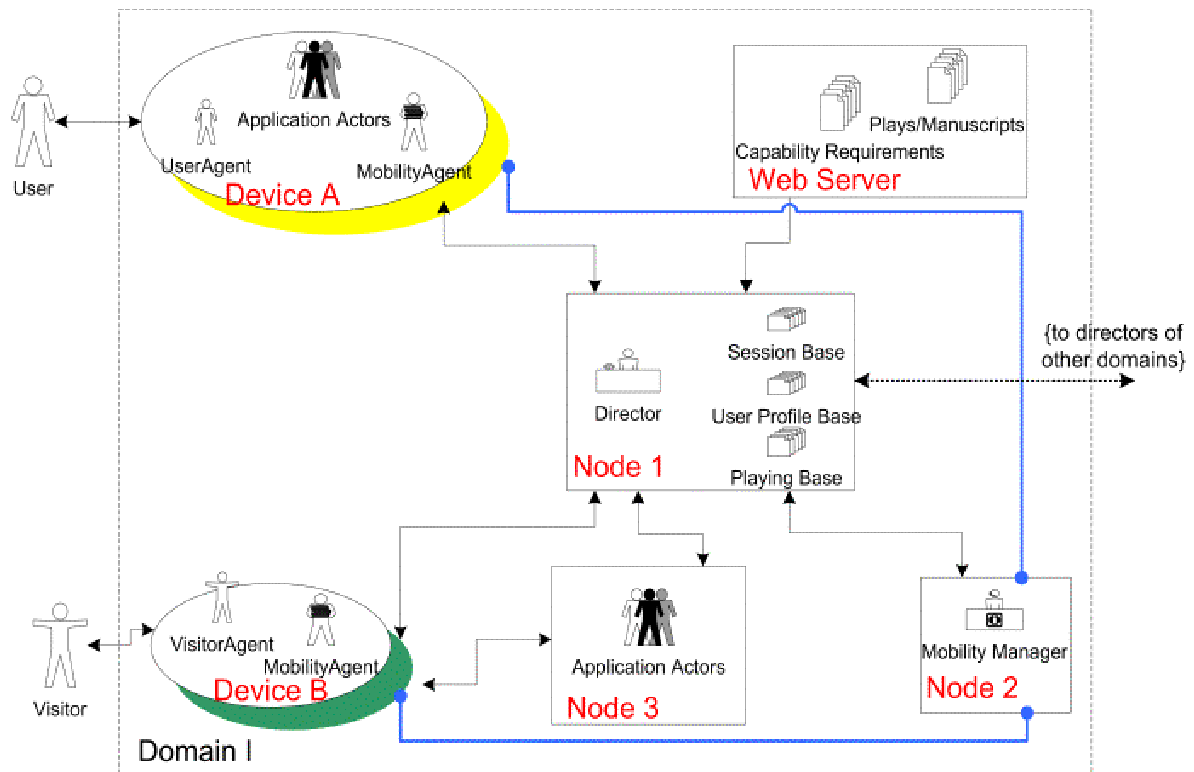

Figure 2.2: General Engineering model for mobility support in the PaP architecture.

2.2 Focus on User and Session Mobility

What is central to our approach is the terminal identification and mobility of sessions and users. Within this framework terms and notions have been defined and related to active entities in the system. For example:

- User is referred to by an ID and profile,
- user-to-device relation is defined at login phase,
- a user-application interaction is controlled by a UserAgent,
- every time a user logs in a user session is maintained,
- personal content is defined by applications and is downloadable,
- applications are dependable on complex capability system regarding device, domain and environment characteristics, etc.

In our approach user's interactions are controlled by a useragent and user sessions are maintained by the director in terms of sessiondescriptions, which are detailed sketches of running applications, actors and their related data. All applications will be carried out through the instantiation of actors with certain functionality. Therefore actors need to be maintained in the user's sessiondescription and/or profile. Figure 2.3 demonstrates how a user's session is managed by the useragent, and consequently maintained by the director's databases. An example is provided for a sessiondescription and a userprofile. In this example application actors are distributed on the user's device and a network node. A typical example is a chatclient and a chatserver, which are managed by the user's agent. When a session is suspended information on every actor's data, e.g. user nickname, online connections with other actors, type of application and additional information for child sessions should be stored. The useragent is completely responsible for the recreation and proper setting of all these application actors.

User's login phase is central to the definition of users identity, characterization of device capabilities, resumption of user sessions, and transfer of personal content. It is used by the director to provide users with proper access rights and profiles. Therefore, there are two types of logins in a multi-domain environment, as shown in figure 2.4. Eventually, users could access their home domain after some inter-director negotiation and authentication. Actors are identified by their GAIs (Global Actor Identifier) and could move between nodes and running processes. Any specific actor that is required by a user's session has to be moved to the user's new location.

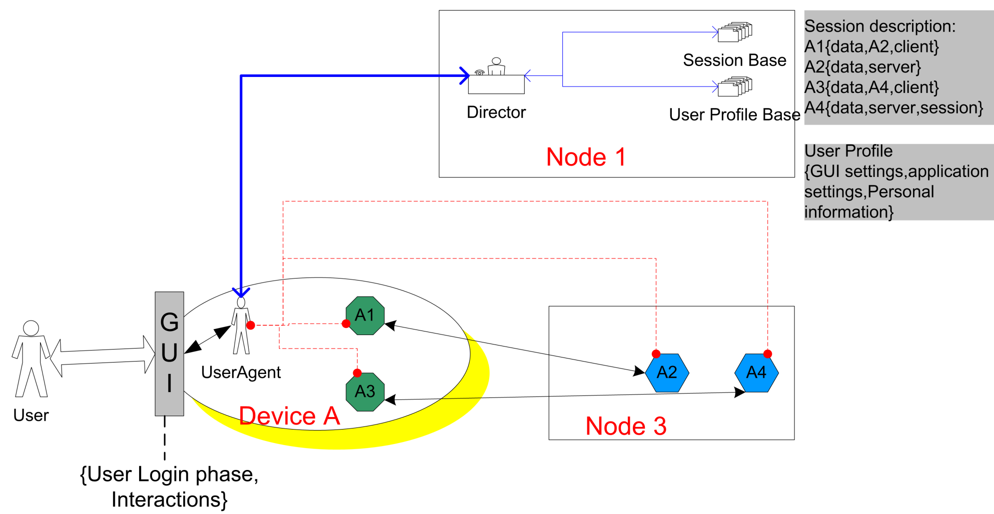

Figure 2.3: Session mobility: A user is interacting with the system through useragent and its session is stored in the director's SessionBase.

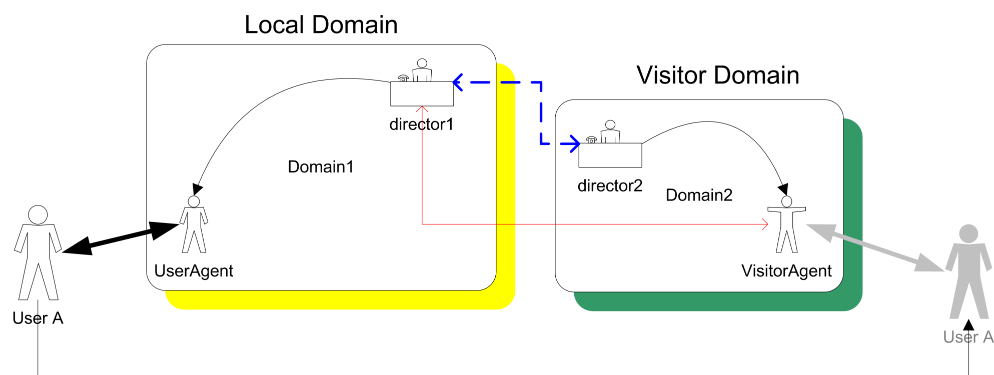

Figure 2.4: User mobility: User A is accessing his home domain from Domain2.

Chapter 3

The mobility architecture

Contents

3.1 Requirements to the mobility framework 24

3.1.1 Functional requirements 24

3.1.2 Non-functional requirements 24

3.2 Use cases 26

3.3 Screenshot 28

3.4 Class diagram 29

3.5 Sequence diagrams 31

3.5.1 Login user 31

3.5.2 Login user remotely 32

3.5.3 Logout user 33

3.5.4 Plugin actor 34

3.5.5 Plugout actor 35

3.5.6 Suspend session 36

3.5.7 Resume session 37

3.5.8 Update session 38

3.1 Requirements to the mobility framework

3.1.1 Functional requirements

The functional requirements for the design of user and session mobility in the Plug-and-Play architecture are stated below. The requirements are based on the assumption that the functionality stated are incorporated into a mobility manager.

1. There must exist a mobility manager in each domain that administrates both the userprofiles of the users belonging to the manager's domain and their sessiondescriptions. A user can only have one userprofile and one sessiondescription and this is administrated by the Mobility manager of the user's home domain.

2. The userprofiles and sessiondescriptions must be persistently stored, for example in one or more textfiles.

3. A user should be able to logon to his mobility manager by sending a loginrequest stating his username and password. The manager will then check if the username is valid and if the password is correct. In the request the user must specify whether to start a new session or to resume a previously suspended session.

4. A user should be able to login as a visitor in a PaP domain by sending a special type of loginrequest to the mobility manager of the domain. He should then get some default set of privileges that are granted to all visitors.

5. A user should be able to logoff from a PaP domain by sending a logoutrequest to the mobility manager of the domain.

6. A user should be able to suspend his current session. The list of actors that the user has plugged in must then be saved, as well as information about the actor's state, in the user's sessiondescription.

7. A user should be able to resume a session that he has previously suspended. How much of the session that can be restored depends on how much information that is stored in the user's sessiondescription.

8. It must be the user's choice whether he wants to store his current session or not.

9. The design should support multiple domains, meaning that a user should get access to both his userprofile and his sessiondescription from a domain other than his home domain. In order for this to happen he must be logged in as a visitor in a domain, send a loginrequest to his current mobility manager who then forwards the request to the mobility manager in his home domain. He will then perform a logon as stated earlier.

3.1.2 Non-functional requirements

The non-functional requirements for the design of user and session mobility in the Plug-and-Play architecture are stated below.

1. Usability - It should be very simple and intuitive to learn how to use the functionality provided by the mobility architecture. It should also be straight forward to setup and configure the PaP system to use the new functionality.

2. Performance - the components that together form the desired functionality must not be so overwhelming and resource demanding that they become a bottleneck in the PaP system.

3. Space - part of the functionality is incorporated into actors (UserAgent, VisitorAgent and LoginAgent) that are supposed to be sent over the network and therefore must be as small as possible to keep the transfer delay at a minimum.

4. Safety - the application must not include functionality that might be able to compromise any security of the network where the application is run.

5. Implementation - the functionality shall be implemented in 100% pure Java. The graphical user interfaces are to be built using Sun's Java Swing classes.

6. Design - the design must be as flexible as possible so that it can easily be extended with new functionality at a later time. Such functionality can be support for terminal and actor mobility.

3.2 Use cases

The use cases for the design of user and session mobility is shown in figure 3.1. The functionality of the mobility manager has been incorporated into the director actor so in effect a director of a domain is acting as the mobility manager of the same domain. As shown in the figure there exists three different agents; loginagent, visitoragent and useragent. They can all communicate with the director but they differ in what type of operations they can perform for a user.

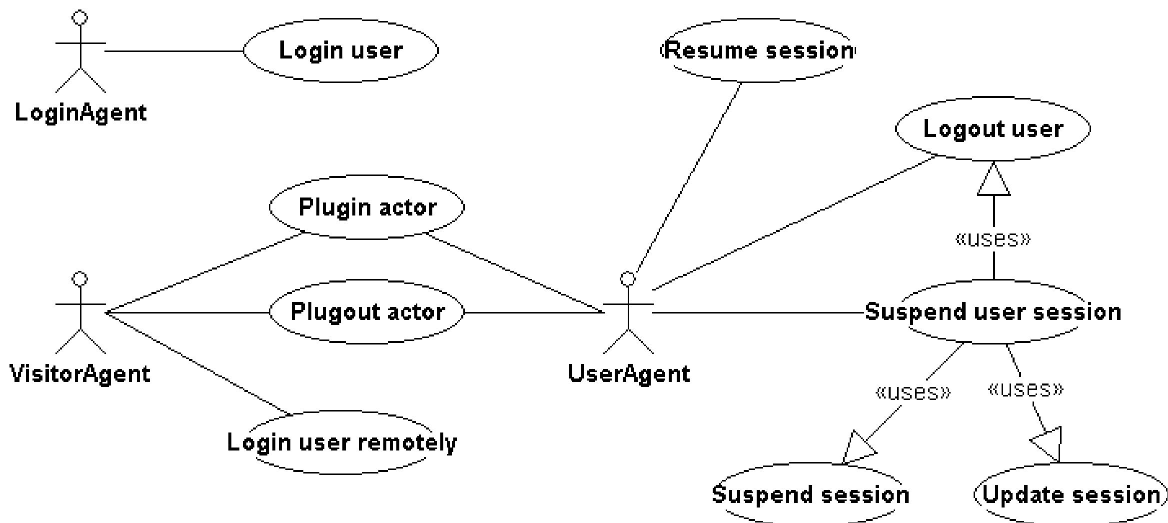

Figure 3.1: Use cases for the design of user and session mobility.

The loginagent is responsible for logging a user into a domain. He sends a loginrequest to the director stating the user's username and password, and whether the user chooses to start a new or resume a previously suspended session.

When a user has logged on as a visitor in a PaP domain a visitoragent is plugged into his node. The visitoragent has a graphical user interface that is acting as an interface to the PaP system. He can plug-in actors with a selected role as well as plug-out one of the actors previously plugged in. When the user chooses to log-off the visitoragent is plugged out, and a new loginagent is provided to the user. Finally, a visitoragent can login a user from another PaP domain to his home PaP domain. This is done by sending a loginrequest to the local director who forwards it to the director of the user's home domain. He will then login the user to his domain. If the logon is successful the visitoragent is plugged out and a useragent is plugged into the user's node. Note that a new PAS is started on the user's node where the PAS' homeinterface is different from the PNES' homeinterface. The PAS' homeinterface is explicitly set to be the GAI of the director of the user's home domain. One can see this as extending the domain of the director to include the new PAS on the user's node. The useragent's director will accordingly be the director of the user's home domain.

A useragent is very similar to a visitoragent since they both are acting as an interface to the PaP system. They differ in what operations they can perform for the user. A user-agent is provided to the user when he has successfully logged into his home PaP domain. He will then get access to his userprofile as well as the chance to resume a previously suspended session. Users logged into a domain as visitors naturally do not have access to neither their userprofile nore their sessiondescription.

A useragent can plug-in and plug-out actors and log off the user from the domain just like a visitoragent. He can also suspend the current session of a user as well as resume a previously suspended session based on the information in a sessiondescription. When the useragent has finished suspending the user's session, the generated sessiondescription, in the form of a Session-object, is sent to the director in a SessionUpdate request. The director will then add the Session-object to his SessionBase. The sessiondescription is persistently stored when the content of the SessionBase is written to an XML-file.

3.3 Screenshot

Figure 3.2 shows a screenshot of the three agents used in the mobility framework. In the top, right corner is the loginagent. In this case user2 has provided his username and password, he has chosen to logon locally and to resume a previously suspended session.

In the top, left corner is the visitoragent and on the bottom is the useragent. The data shown in the tables are the content of the file RoleList.xml. See section 4.2 for more information about this file. The visitoragent only shows four entries while the useragent shows five. This is because in the file RoleList.xml the chatserver actor has the property ForVisitors set to false. The visitoragent only shows the entries where this property is set to true. A user can logoff from a PaP domain by either selecting the [Log out] menuitem or selecting the [Suspend session] menuitem. The latter makes the useragent suspend the user's session before he is logged off.

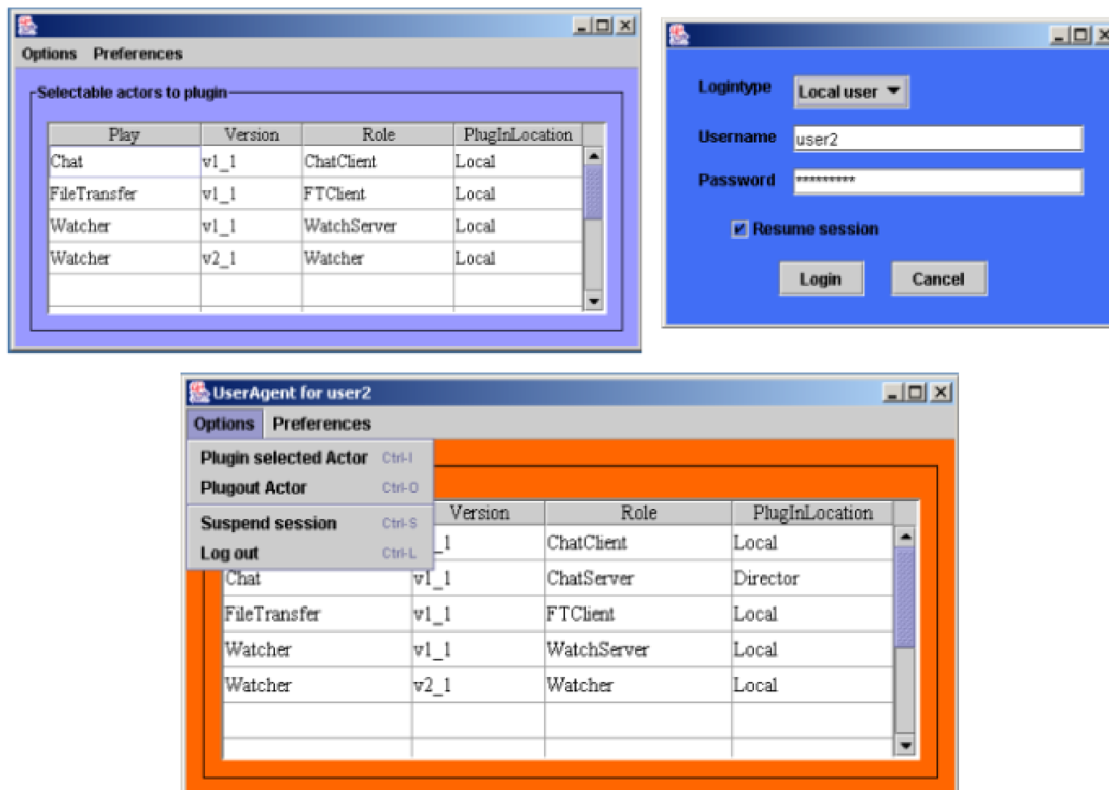

Figure 3.2: Screenshot of the agents used in the mobility framework.

3.4 Class diagram

Figure 3.3 shows the class diagram of the mobility framework. The classes shown in the diagram are only the classes belonging to the play called User. This play must be plugged in in order to use the mobility framework. Other classes introduced through the framework, such as Session, SessionBase and LoginRequest, are added to the package PaP.

The classes for the three agents used in the framework are LoginAgent, VisitorAgent and UserAgent. They inherit the ApplicationActor class like every other actor. All three classes have their own GUI class; LoginWindow, VisitorWindow and UserWindow. The classes UserAgent and VisitorAgent are almost identical since they provide similar functionality to the user. They both have a reference to an instance of XMLFileUtil. It is used to read the file RoleList.xml described in section 4.2. In addition the UserAgent uses XMLFileUtil for some tasks used to resume and suspend a user's session.

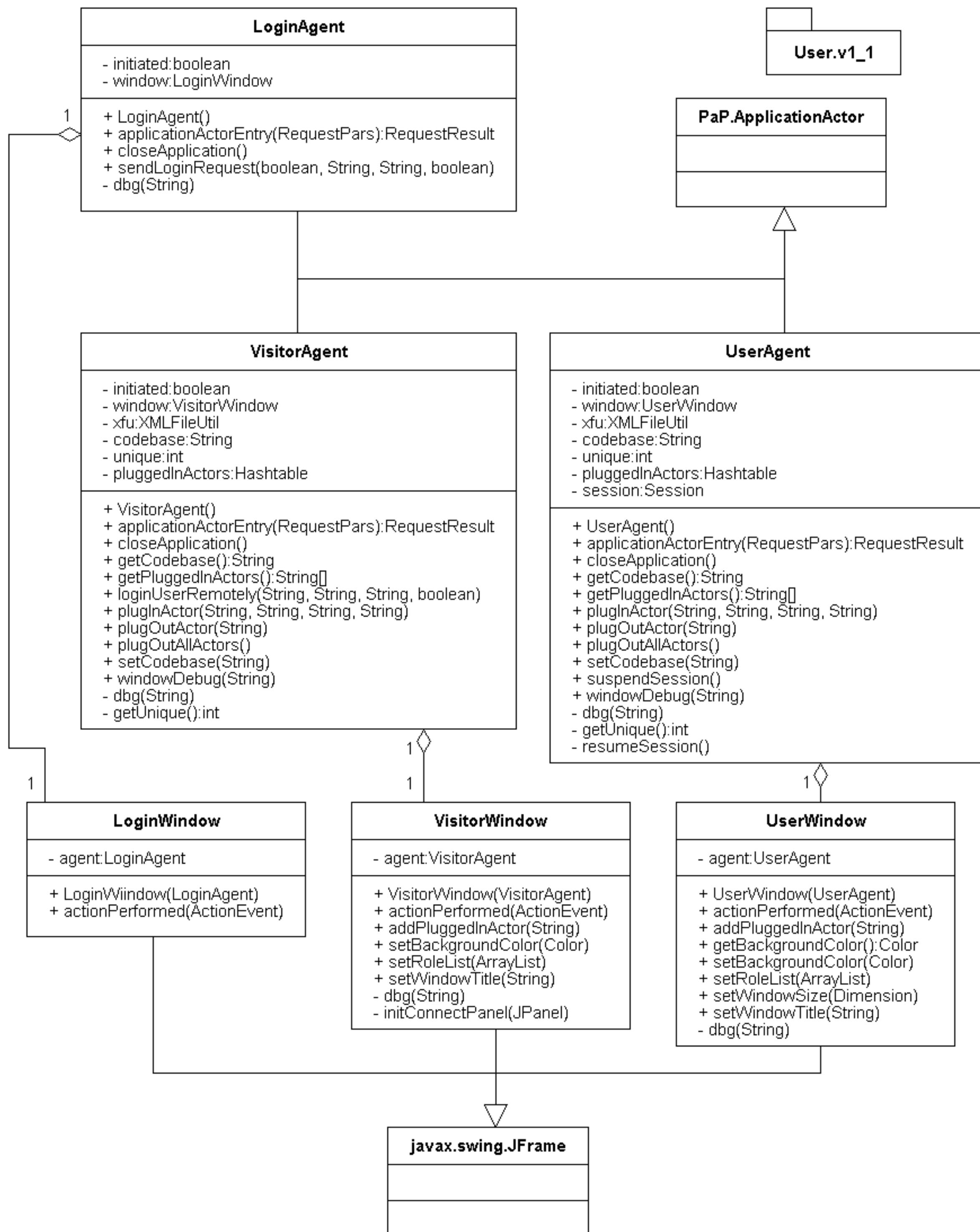

Figure 3.3: Class diagram for the mobility framework.

3.5 Sequence diagrams

3.5.1 Login user

Sequence diagram for the Login user use case is shown in figure 3.4. When a user wants to logon to a PaP domain his loginagent sends a logonrequest to the director of the domain. If the logonrequest is of type LOCAL and the logon is successful, a useragent is provided to the user. In the request he specifies whether to resume a previously suspended session or start a new one. If the logonrequest is of type VISITOR and the logon is successful, a visitoragent is provided to the user.

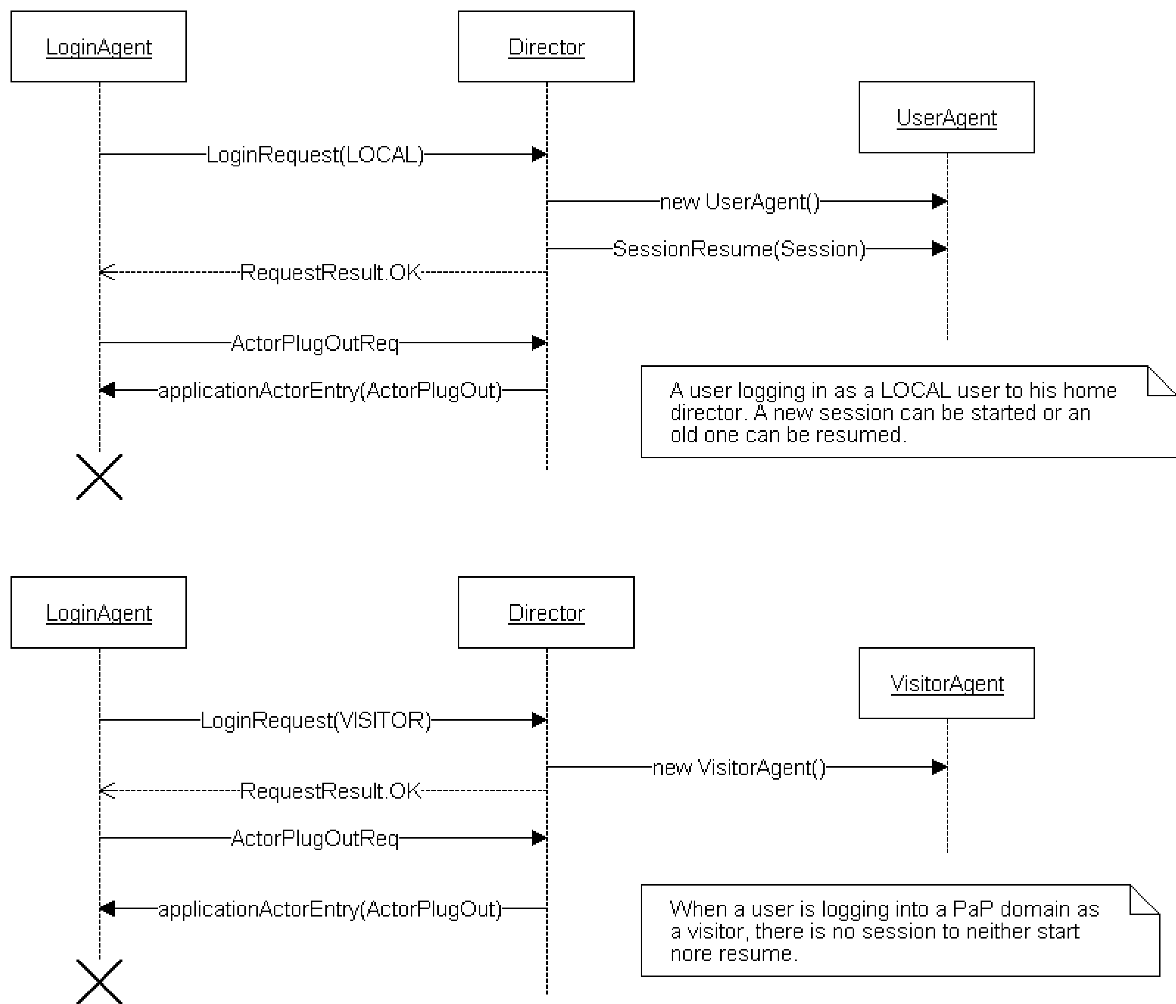

Figure 3.4: Sequence diagram for logging in a user to PaP domain.

3.5.2 Login user remotely

Sequence diagram for the Login user remotely use case is shown in figure 3.5. When a user is logged on as a visitor he can later logon to his home PaP domain. This is done by sending a logonrequest of type REMOTE to his current director. The logonrequest contains the Global Actor Identifier of the user's home director. The visitor director forwards the request to the home director who logs the user onto his domain. If the logon was successful a useragent is plugged into the user's node and the visitoragent is plugged out. Note that the visitoragent has the visitor director as his homeinterface while the useragent has the home director as his homeinterface. One might think of this as the home director expanding his domain to include part of the user's node.

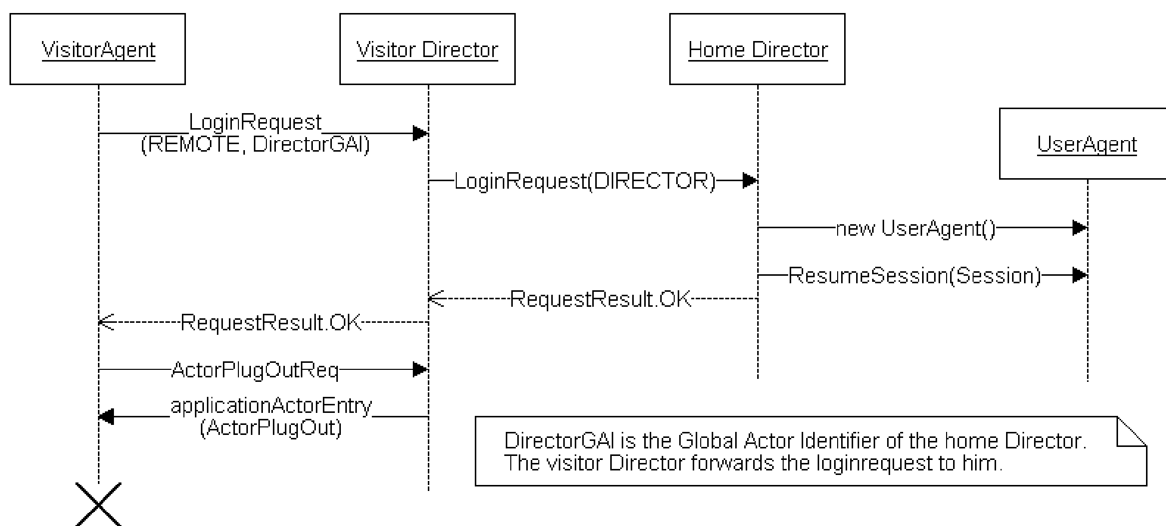

Figure 3.5: Sequence diagram for logging in a user from another PaP domain.

3.5.3 Logout user

Sequence diagram for the Logout user use case is shown in figure 3.6. When a user wants to logoff from a PaP domain the useragent sends a logoutrequest to the director. The useragent adds the user's username to the logoutrequest so that the director can set the loggedIn property of the user's userprofile to false. If the user is successfully logged out from the domain, a new loginagent is plugged into the user's node before the useragent plugs himself out.

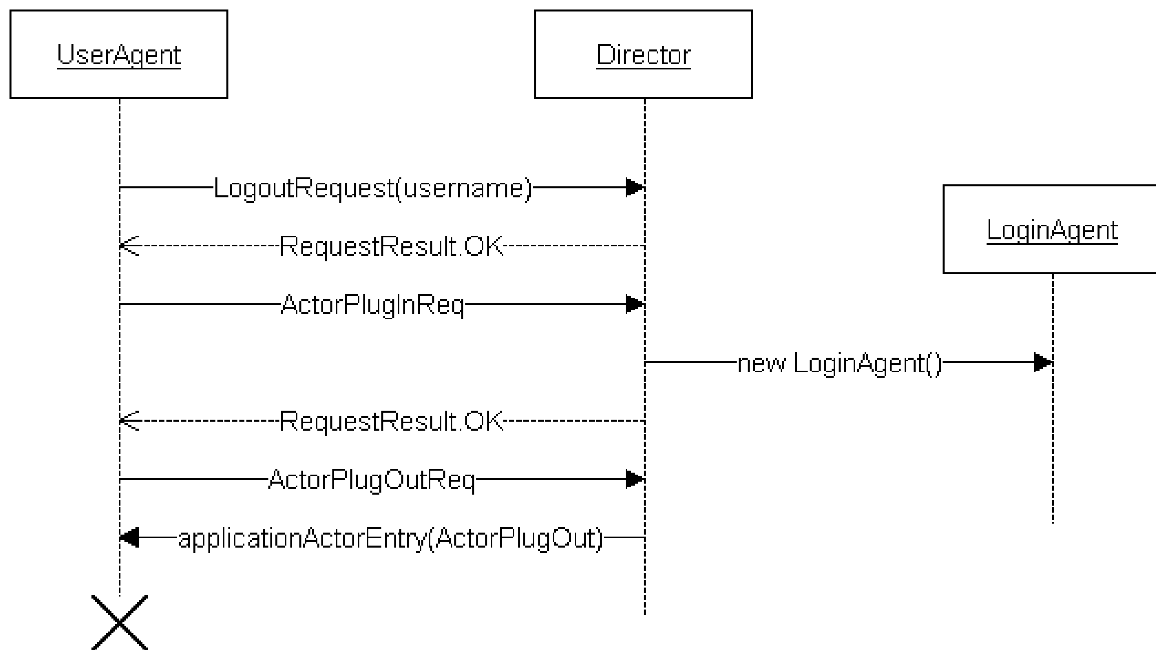

Figure 3.6: Sequence diagram for logging out a user from a PaP domain.

3.5.4 Plugin actor

Sequence diagram for the Plugin actor use case is shown in figure 3.7. When a user wants to plug-in an actor he selects the actor's role from the list shown in the userwindow. Userwindow is the graphical user interface of the useragent. The userwindow calls the method pluginActor() in the useragent specifying the play to plugin and the role to assigned to the new actor. The useragent sends a PlayPlugIn request to the director requesting to plug-in the specified play. Then an ActorPlugIn request is sent to the director requesting to plug-in an actor with the specified role. Depending on the location parameter the new actor is either plugged in on the user's node or on the director's node.

In the figure below the actor is called ApplicationActor since all actors inherit the class ApplicationActor. Note that no matter if the PlayPlugIn request succeeds or not the ActorPlugIn request is sent. The reason for this is that if the request fails it could either mean that something was wrong or that the play was already plugged in. There is no way to tell. An actor is plugged in the same way by the visitoragent except that then the visitorwindow is calling the method pluginActor() in the visitoragent.

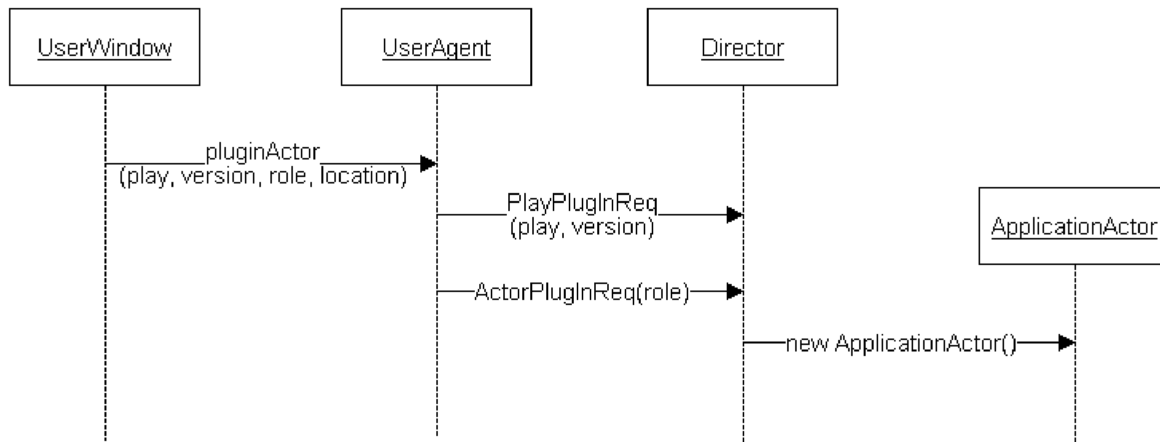

Figure 3.7: Sequence diagram for plugging in an actor.

3.5.5 Plugout actor

Sequence diagram for the Plugout actor use case is shown in figure 3.8. When a user wants to plug-out an actor previously plugged in he selects the actor's Global Actor Identifier from the list in the userwindow. Userwindow is the graphical user interface of the useragent. The userwindow calls the method actorPlugout() in the useragent specifying the actor to plug-out. The useragent sends an ActorPlugOut request to the director who pluggs out the specified actor. In the figure below the actor is called ApplicationActor since all actors inherit the class ApplicationActor. An actor is plugged out the same way by the visitoragent except that then the visitorwindow is calling the the method plugoutActor() in the visitoragent.

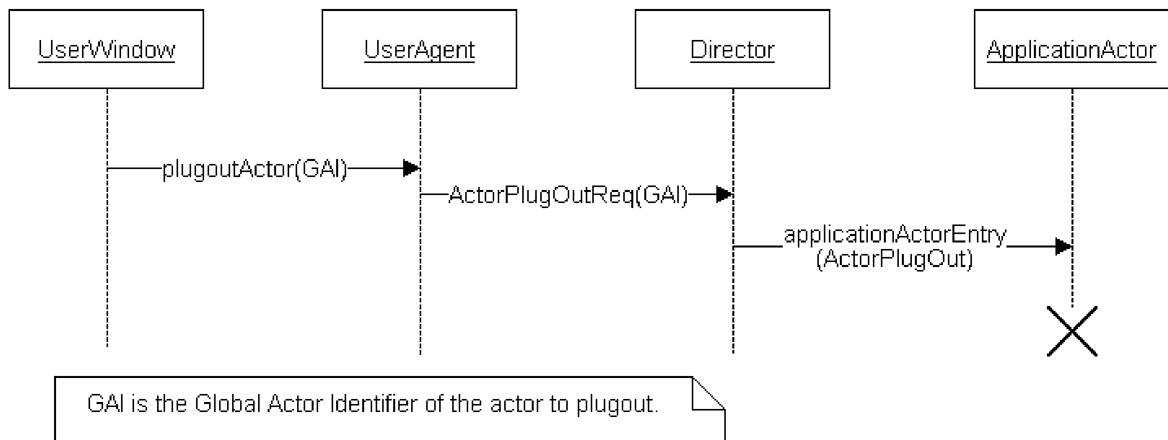

Figure 3.8: Sequence diagram for plugging out an actor.

3.5.6 Suspend session

Sequence diagram for the Suspend session use case is shown in figure 3.9. The useragent contains a list of all the actors that he has previously plugged in. In order to get a complete sessiondescription one must, among other things, gather the state of all these actors as well as their rolesessions. When the user chooses to suspend his current session the useragent in turn sends a SessionSuspend request to each of these actors. They return their current state which consists of a set of variables mentioned in section 4.3 as well as a list of the actor's rolesessions. Note that it is implementation specific how much information and how many of his rolesessions an actor wants to save.

The information returned to the useragent is added to his local copy of the user's sessiondescription. Other information gathered about an actor is the role he is playing, the name and the version of the play that the role belongs to and the name of the actor. After the actors have returned their state they are plugged out.

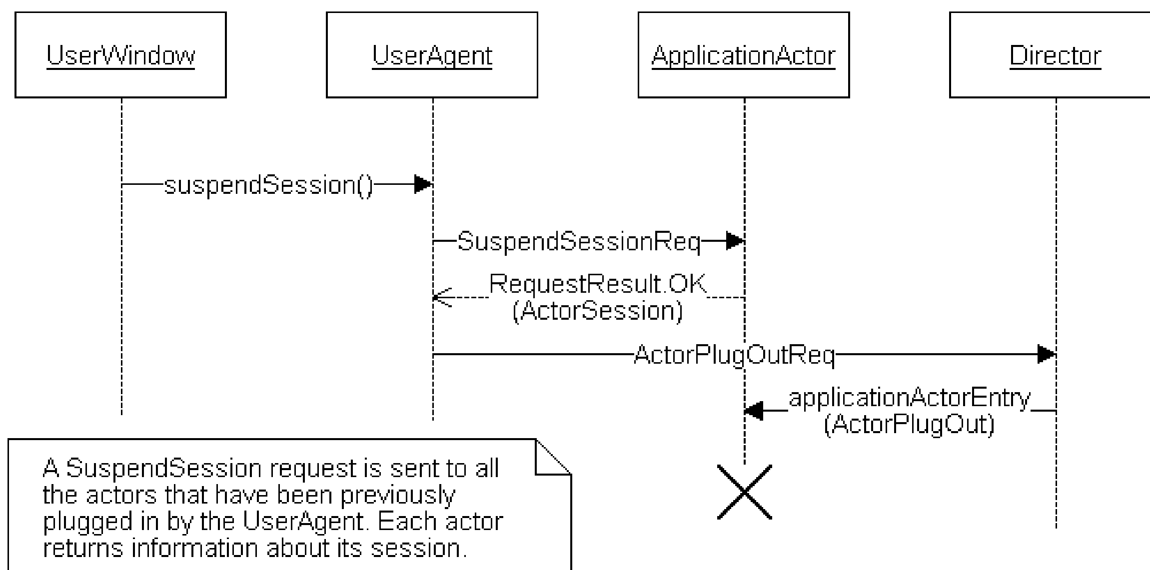

Figure 3.9: Sequence diagram for suspending a user's session.

3.5.7 Resume session

Sequence diagram for the Resume session use case is shown in figure 3.10. After the useragent is plugged in the director sends it a SessionResume request with the user's sessiondescription, in the form of a Session-object, added to it. If the user has chosen not to resume a previously suspended session, the Session-object is empty and does not contain any actorinstances to recreate. One can think of this as resuming an empty session. If the user has chosen to resume his session, the Session-object contains a list of actorinstances. An actorinstance is an ArrayList with information about a suspended actor such as its name, role and state.

An actor is recreated by first plugging in the play that the actor's role belongs to. Then an actor is plugged in with the specified role. Further, the actor's rolesessions are recreated by sending RoleFigurePlugIn requests to the director. If the other participant of a rolesession is unavailable or non-existing, the rolesession can not be recreated. Finally, the actor is set to the state it was in when he was suspended. This is done by sending him a SessionResume request with the actor's state information added to it. These operations are performed for each actorinstance in the user's Session-object.

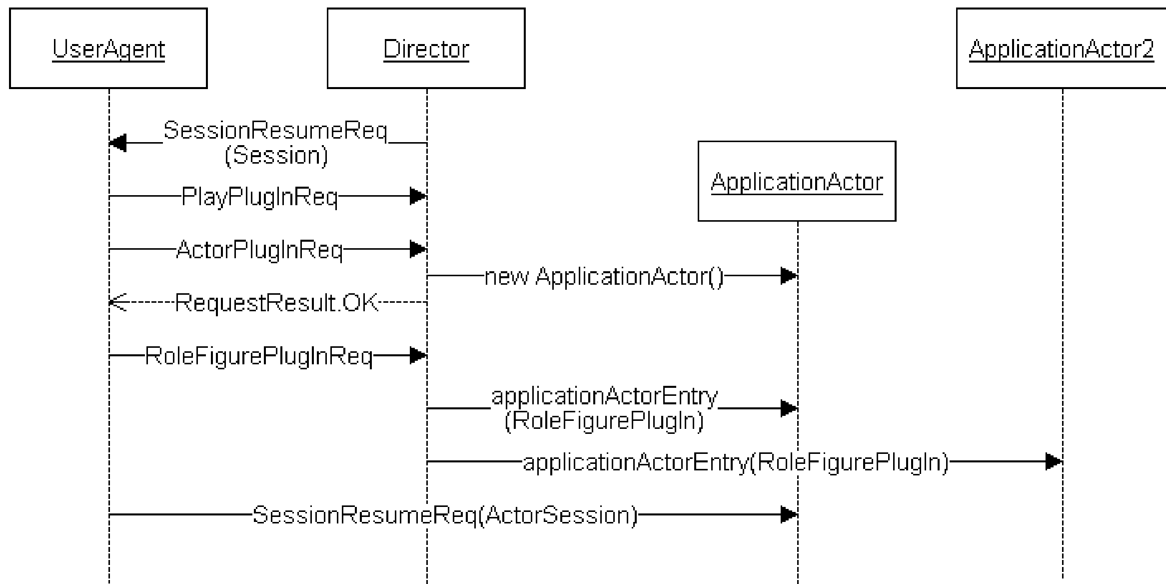

Figure 3.10: Sequence diagram for resuming a user's session.

3.5.8 Update session

Sequence diagram for the Update session use case is shown in figure 3.11. When a user has chosen to suspend his current session the useragent collects information and adds it to his Session-object. In order for this information to be stored persistently the useragent must send a SessionUpdate request to the director containing the updated Session-object. The director then updates the SessionBase with the new Session-object. The SessionBase is where all the Session-objects of the users of the PaP domain is stored. Each time a SessionUpdate is received, the content of the SessionBase is written to the file SessionDescriptions.xml. See section 4.3 for more information about this file.

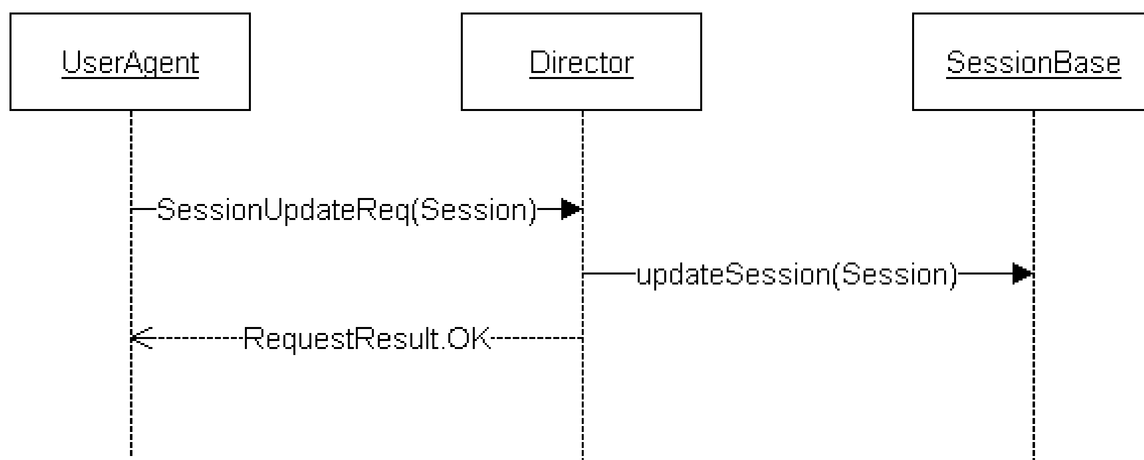

Figure 3.11: Sequence diagram for updating a user's session.

Chapter 4

The XML files

Contents

4.1 What is their use **40**
4.1.1 Why do we need to store information 40
4.1.2 What is XML and why was it chosen 40
4.2 The file RoleList.xml **41**
4.3 The file SessionDescriptions.xml **42**
4.4 The file UserProfiles.xml **44**

4.1 What is their use

4.1.1 Why do we need to store information

For the functionality provided by the mobility architecture to work, some pieces of information must be provided when the PaP system is initialized. It would be very inconvenient to have to provide this manually every time the system is started. The information has therefore been put into a set of text-files which is read and parsed when needed. Also, the functionality of the mobility framework is better utilized when a user can both logoff from a PaP domain and restart the computer without losing his userprofile and sessiondescription. Finally, by persistently storing the information needed the PaP system is made more robust and failsafe.

4.1.2 What is XML and why was it chosen

XML stands for eXtensible Markup Language. It has quickly become the universal standard way of storing and distributing information in the software industry. An XML-file contains a set of tags where < and > marks the start and end of a tag, respectively. Tags always appear in pairs where one is a starttag and another is an endtag. The difference between them is that the endtag has a / before the tagname. For example the string ”<a>value” contains two tags where <a> is the starttag and is the endtag. Everything between a starttag and an endtag is called its value. Tags can also be nested so that the value is a set of tags, as is shown later in the chapter.

The amount of information to be stored is fairly limited so to use a database for this task would be overkill. In addition it would require one to have a database system in order to run the PaP system, which is not ideal. Instead one chose to store the information in text-files, and XML was chosen as the syntax of the files because the language is very flexible which makes it simple to change the grammar of a file. In addtion it is very easy to parse the content of an XML-file. The files described in this chapter have an XML inspired syntax and the content is so called valid. See [9] for more information about the XML standard.

4.2 The file RoleList.xml

The file RoleList.xml specifies which roles a visitoragent and useragent can assign to new actors. It is read by either of the two types of actors when they are plugged in. The tags <RoleList> and </RoleList> mark the start and end of the rolelist, respectively. The list contains a set of roles and the name and version of the play that the role belongs to. The tags PlugInLocation specify whether the new actor shall be plugged in on the user's node (Local) or on the director's node (Director). The tags ForVisitors specify whether a user logged in as a visitor can plug-in an actor with the specified role (Yes/No). By including this property the same file can be used by both a visitoragent and a useragent, but their interpretation of the file is different. Basically, when a role is selected the play the role belongs to is plugged in. Then an actor is plugged in with the selected role. An example of the syntax of the file is shown below.

```
<RoleList>
   <Role>
      <Play>Chat</Play>
      <Version>v1_1</Version>
      <RoleName>ChatClient</RoleName>
      <PlugInLocation>Local</PlugInLocation>
      <ForVisitors>Yes</ForVisitors>
   </Role>
   <Role>
      <Play>Chat</Play>
      <Version>v1_1</Version>
      <Name>ChatServer</Name>
      <PlugInLocation>Director</PlugInLocation>
      <ForVisitors>No</ForVisitors>
   </Role>
</RoleList>
```

4.3 The file SessionDescriptions.xml

The file SessionDescriptions.xml is used to store the sessiondescriptions of the users of a PaP domain. A sessiondescription contains information about a session that a user previously has suspended. Ideally a sessiondescription should contain enough information so that the whole session can later be resumed. The tags <SessionDescriptions> and </SessionDescriptions> mark the start and end of the file, while the tags <Session> and </Session> mark the start and end of a sessiondescription, respectively. The User tags specify the username of the user that the sessiondescription belongs to. This must be the same username as is written in the user's userprofile.

A sessiondescription contains zero or more actorinstances. An actorinstance is a list of information about an actor that the user has plugged in. The ActorRole tags specify the role of the actor, while the Play and Version tags specify which play the actor's role belongs to. Further, the ActorName tags contains the name of the actor and the ActorState tags specify the actor's state when the session was resumed. The state is a set of variables with their values. It is of the form [variable_name]=[value]. The variables are separated by whitespace.

An example of the syntax of the file is shown below. In the example the first sessiondescription contains an actor named ChatClient1. Its state contains one variable called Username with the value LarsErik. The state information is not standardized and it is implementation specific if an actor wishes to save some information when he is suspended. If no such information is provided the ActorState tags have no value, as shown in the second sessiondescription below.

An actorinstance also contains a list of the actor's rolesessions. Note that this list can be empty. An empty list is omitted from a sessiondescription, which has been done in the second sessiondescription in the example below. The Initiator and Cooperator tags naturally specify the initiator and cooperator of a rolesession. In the example below the Initiator tags have the value this. It marks that it is the current actor, specified by the ActorName tags, that is the initiator of the rolesession. When the session is resumed the value is replaced by the actor's new GAI.

```
<SessionDescriptions>
  <Session>
    <User>user2</User>
    <ActorInstance>
      <Play>Chat</Play>
      <Version>v1_1</Version>
      <ActorRole>ChatClient</ActorRole>
      <ActorName>ChatClient1</ActorName>
      <RoleSession>
        <Initiator>this</Initiator>
        <Cooperator>Actor://1.1.1.1/pas1/ChatServer1</Cooperator>
      </RoleSession>
      <ActorState>Username=LarsErik</ActorState>
    </ActorInstance>
  </Session>

  <Session>
    <User>user4</User>
    <ActorInstance>
      <Play>Watcher</Play>
      <Version>v1_1</Version>
      <ActorRole>Watcher</ActorRole>
      <ActorName>W1</ActorName>
      <ActorState></ActorState>
    </ActorInstance>
  </Session>
<SessionDescriptions>
```

4.4 The file UserProfiles.xml

The file UserProfiles.xml is used to store the userprofiles of the users of a PaP domain. At the moment a userprofile only contains the username and password for a user, and a set of properties used to visually show user mobility. In the future more fields, such as access rights, are likely to be added.

The tags <UserProfiles> and </UserProfiles> mark the start and end of the file while the tags <User> and </User> mark the start and end of a userprofile, respectively. The BackgroundColor tags specify the background color of the userwindow. It uses the RGB (Red-Green-Blue) color model where the value of the three colors are listed with a white-space between them. The WindowLocation tags specify the top-left corner of the user-window. The numbers mark the x and y value of the corner, respectively. Finally, the WindowSize tags specify the size of the userwindow. The numbers mark the width and height of the window, respectively. An example of the syntax of the file is shown below.

```
<UserProfiles>
   <User>
      <Username>user1</Username>
      <Password>password1</Password>
      <BackgroundColor>153 255 153</BackgroundColor>
      <WindowLocation>326 77</WindowLocation>
      <WindowSize>645 371</WindowSize>
   </User>
   <User>
      <Username>user2</Username>
      <Password>password2</Password>
      <BackgroundColor>0 255 0</BackgroundColor>
      <WindowLocation>100 100</WindowLocation>
      <WindowSize>350 350</WindowSize>
   </User>
</UserProfiles>
```

Chapter 5

The Chat application

Contents

5.1 Requirements to the Chat application 46

5.1.1 Functional requirements 46

5.1.2 Non-functional requirements 47

5.2 Use cases 48

5.3 Screenshot 49

5.4 Class diagram 50

5.5 Sequence diagrams 51

5.5.1 Connect 51

5.5.2 Disconnect 52

5.5.3 Get userlist 53

5.5.4 Send and receive message 54

5.5.5 Suspend session 55

5.5.6 Resume session 56

5.1 Requirements to the Chat application

5.1.1 Functional requirements

The functional requirements for the Chat application are stated below.

1. A chatclient should be able to connect to a chatserver by using a specified username. This username must be unique within the context of the chatserver. This means that two users can not connect to the same chatserver with the same username.

2. A chatclient should be able to disconnect from the chatserver he is currently connected to.

3. A chatclient should have a graphical user interface that can show the user a log of all previously sent and received messages.

4. A user should be able to write and send new messages to a chatserver. The chatserver should then be able to distribute the messages to all currently connected chatclients.

5. The chatserver should be able to provide to a chatclient a list of all the currently connected chatclients. The list could for example be shown in the chatclient's graphical user interface.

6. A chatclient should be able to suspend his session if a SessionSuspend request is received. The minimum amount of information that should be stored is the chatclient's username and the Global Actor Identifier (GAI) of the chatserver.

7. A chatclient should be able to resume a session that he has previously suspended if a SessionResume request is received. The information needed to resume the session must then be provided in the request.

8. The Chat application should support multiple PaP domains, meaning that a chatclient in one domain should be able to connect to a chatserver in another domain.

5.1.2 Non-functional requirements

The non-functional requirements for the Chat application are stated below.

1. Usability - The Chat application should be intuitive and easy to use.
2. Performance - The components that together form the application must not be so overwhelming and resource demanding that they become a bottleneck in the PaP system.
3. Space - Due to the fact that the components is going to be sent over the network the implementation should be as small and compact as possible to keep the transfer delay at a minimum.
4. Safety - The application must not include functionality that might be able to compromise any security of the network where the application is run.
5. Implementation - The functionality shall be implemented in 100% pure Java. The graphical user interface is to be built using Sun's Java Swing classes.
6. Design - The design must be as flexible as possible so that it can easily be extended with new functionality at a later time.

5.2 Use cases

The use cases for the design of the Chat application is shown in figure 5.1. The play is called Chat and consists of two roles; chatclient and chatserver. Each chatclient has a chatclientWindow which is the graphical user interface through which a user can write new messages and see a log of previously sent and received messages. A chatclient can connect to a chatserver by specifying a unique username. This means that two users can not connect to the same chatserver by using the same username. A user's username can be set through the chatclientWindow and does not have to be equal to the username used to log into the PaP domain.

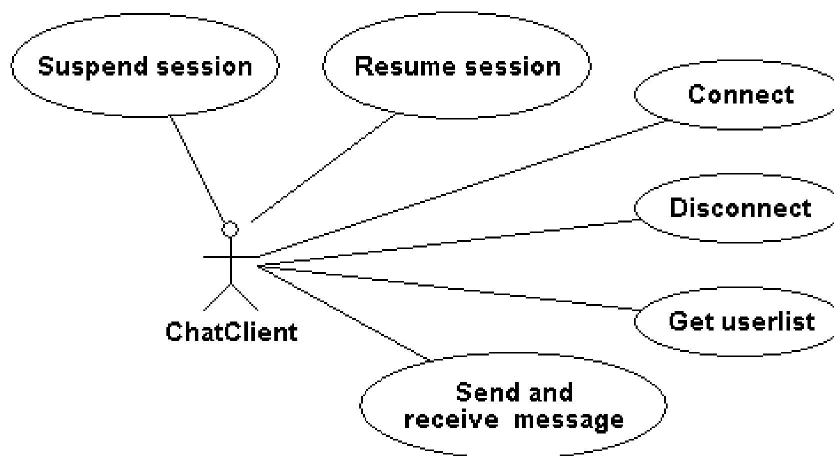

Figure 5.1: Use cases for the design of the Chat application.

A nice feature is the possibility to receive a list of all the chatclients connected to a chatserver. The list consists of the username and the Global Actor Identifier of all the connected chatclients. Finally, a chatclient is able to suspend his current session as well as resume it. The chatserver does not have this ability. Since it is independent of any chatclient it would be meaningless to suspend it when a chatclient is suspended. The fact that chatserver actors are plugged in on the director's node, while chatclients are plugged in on the user's node, emphasizes this.

5.3 Screenshot

Figure 5.2 shows a screenshot of the Chat application. One chatserver actor and two chatclient actors have been plugged in. The username of each of the chatclients has been set and they are both connected to the chatserver. We can see that the chatclient on the right (Lars Erik) connected first, since it has received a message from the chatserver saying that the other chatclient (Liljebäck) was connected.

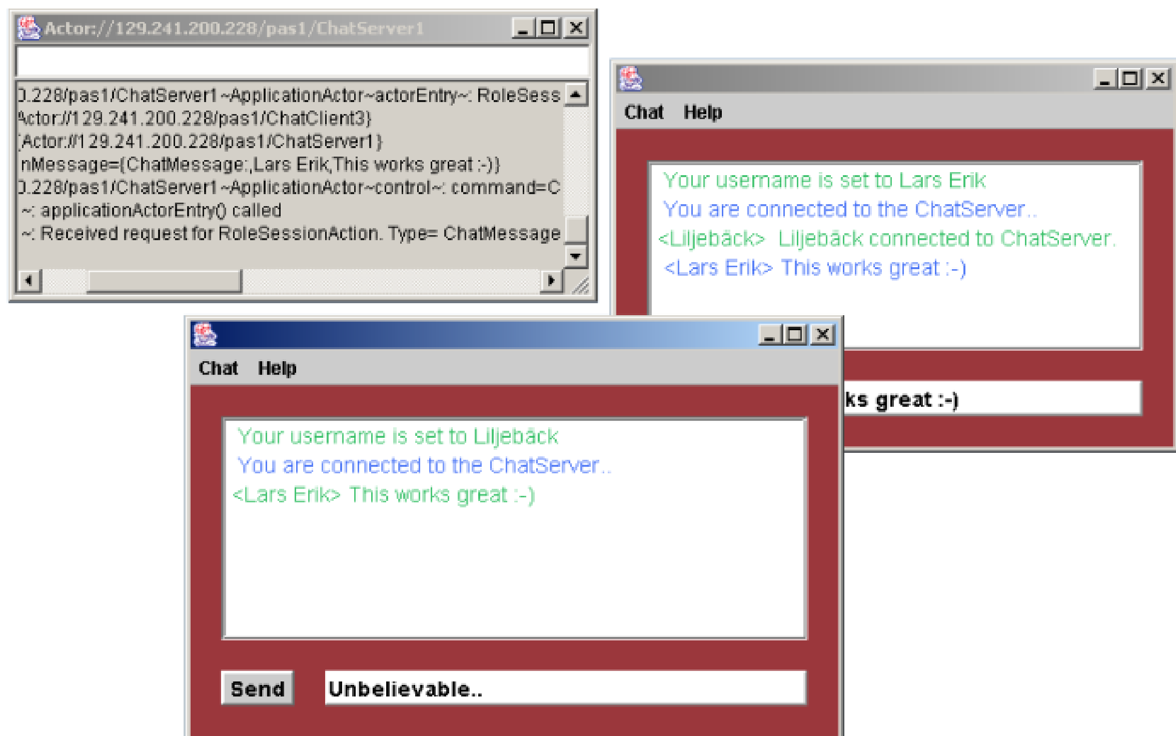

Figure 5.2: Screenshot of the Chat application.

5.4 Class diagram

Figure 5.3 shows the class diagram of the Chat application. Both the classes ChatServer and ChatClient extends the ApplicationActor class. This is a superclass that all actors, except the director actor, must extend. Through inheritance they are able to use the functionality provided by the PaP support system. The ChatClientWindow class constitutes the graphical user interface (GUI) of the chatclient actor. The chatserver actor has no GUI since all the interaction he participates in is done by sending and receiving from chatclients and therefore performs no interaction with the user. The ChatClient and the ChatServer classes contain a number of constants defining the different applicationmessage types. They are used to distinguish messages sent as RoleSessionAction requests.

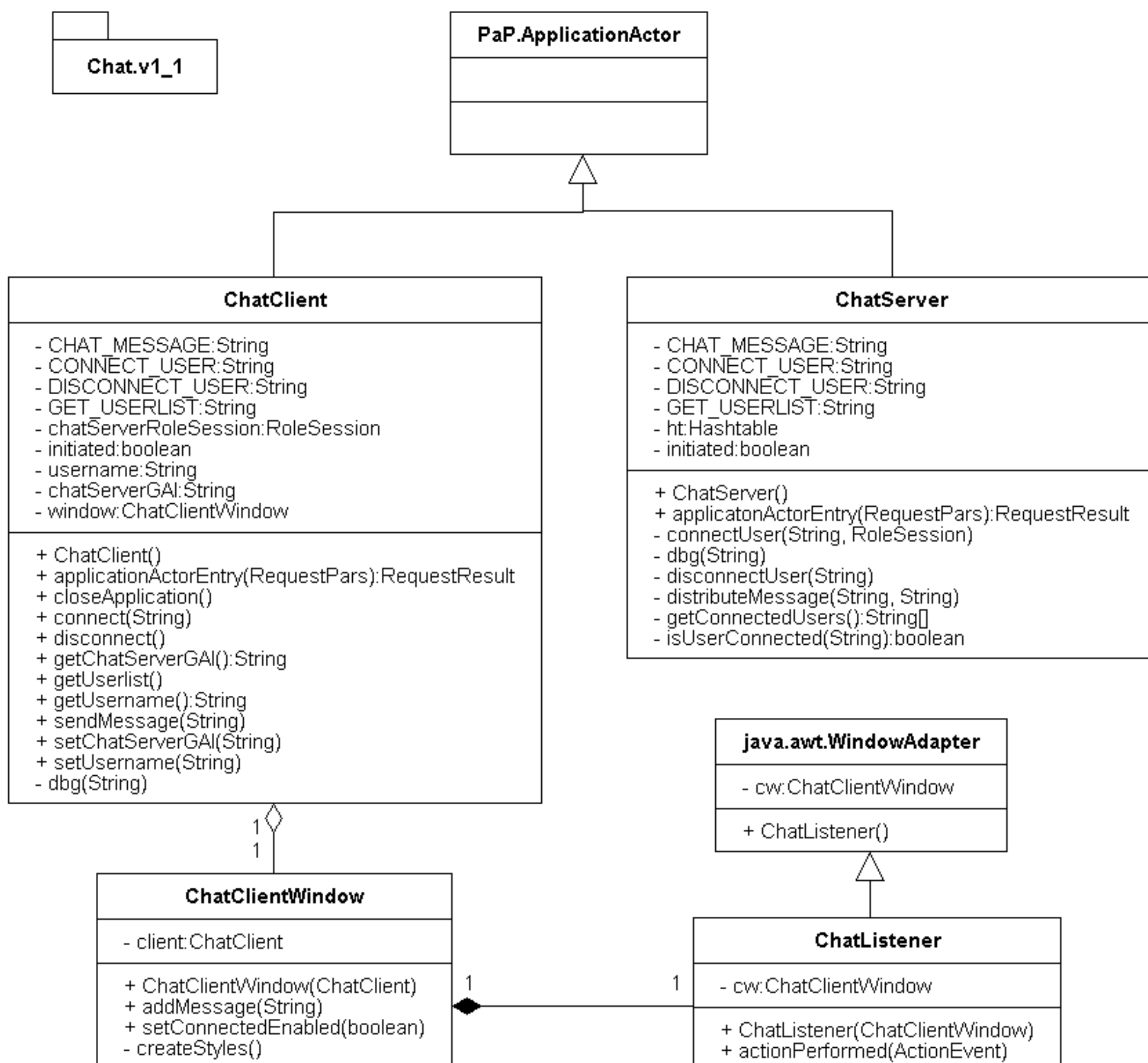

Figure 5.3: Class diagram for the Chat application.

5.5 Sequence diagrams

5.5.1 Connect

Sequence diagram for the Connect use case is shown in figure 5.4. In order for a chatclient to connect to a chatserver he must go through two steps; create a connection to the chatserver and send him a message saying that he wants to connect. The connection is a rolesession.

The chatclient first sends a RoleFigurePlugIn request to the director who creates a rolesession where the chatclient is the initiator and the chatserver is the cooperator. Then the RoleFigurePlugIn request with the new rolesession added, is sent to both actors who will add the new rolesession to their RoleSessionCollections.

After the rolesession has been created the chatclient sends an applicationmessage of type CONNECT_USER to the chatserver specifying his username. In order to notify the other connected chatclients of the new chatclient, an applicationmessage of type CHAT_MESSAGE is sent to each of them specifying the username of the new chatclient.

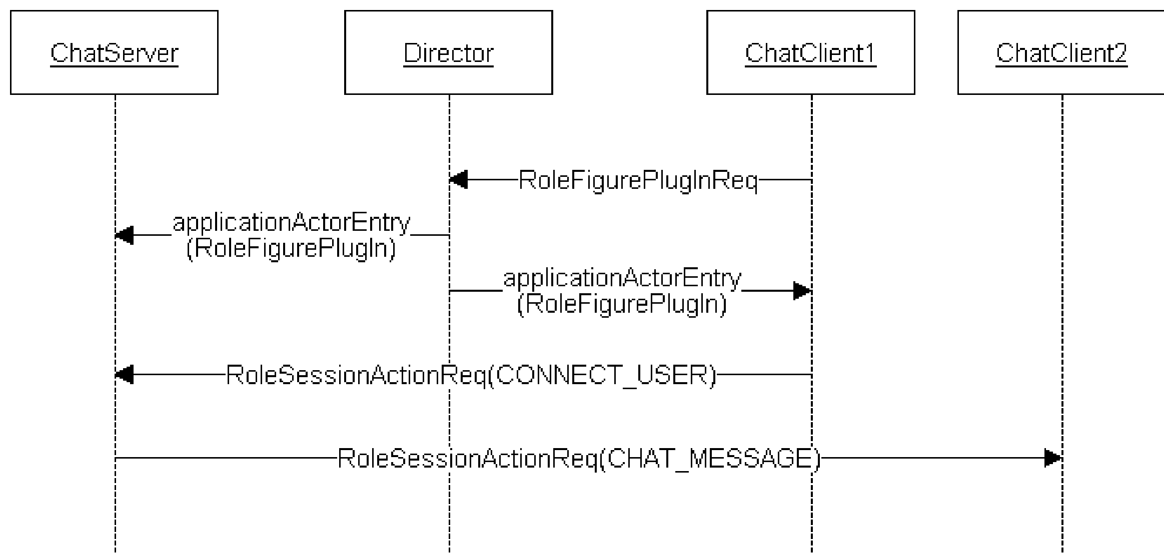

Figure 5.4: Sequence diagram for connecting a chatclient to a chatserver.

5.5.2 Disconnect

Sequence diagram for the Disconnect use case is shown in figure 5.5. When a chatclient wants to disconnect from a chatserver, he needs to reverse the two steps performed when the chatclient connected; send a message to the chatserver saying he wants to disconnect and remove the connection to the chatserver. This is done by first sending an applicationmessage of type DISCONNECT_USER to the chatserver. The chatserver then notifies the other currently connected chatclients by sending them an applicationmessage of type CHAT_MESSAGE specifying which chatclient that disconnected.

The rolesession is removed by sending a RoleFigurePlugOut request to the Director. He forwards the request to both the chatclient and the chatserver who removes the specified rolesession from their RoleSessionCollections.

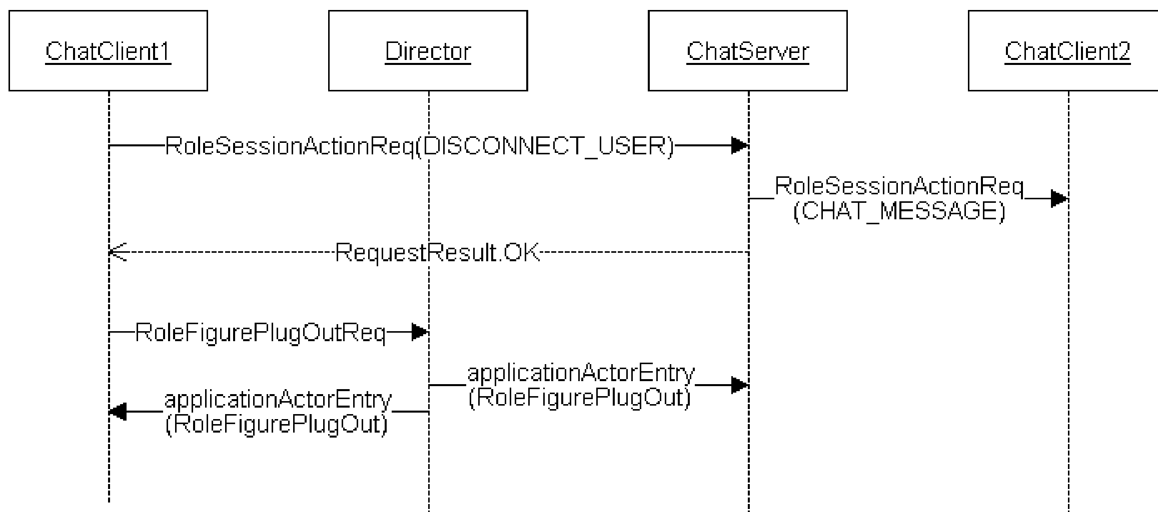

Figure 5.5: Sequence diagram for disconnecting a chatclient from a chatserver.

5.5.3 Get userlist

Sequence diagram for the Get userlist use case is shown in figure 5.6. A chatclient can obtain a list of the connected chatclients to a chatserver by sending an applicationmessage of type GET_USERLIST to the chatserver. He will then generate the list and return a RequestResult with the list added. The list contains the username and the Global Actor Identifier of all connected chatclients, and it is shown in the log in the chatclientwindow.

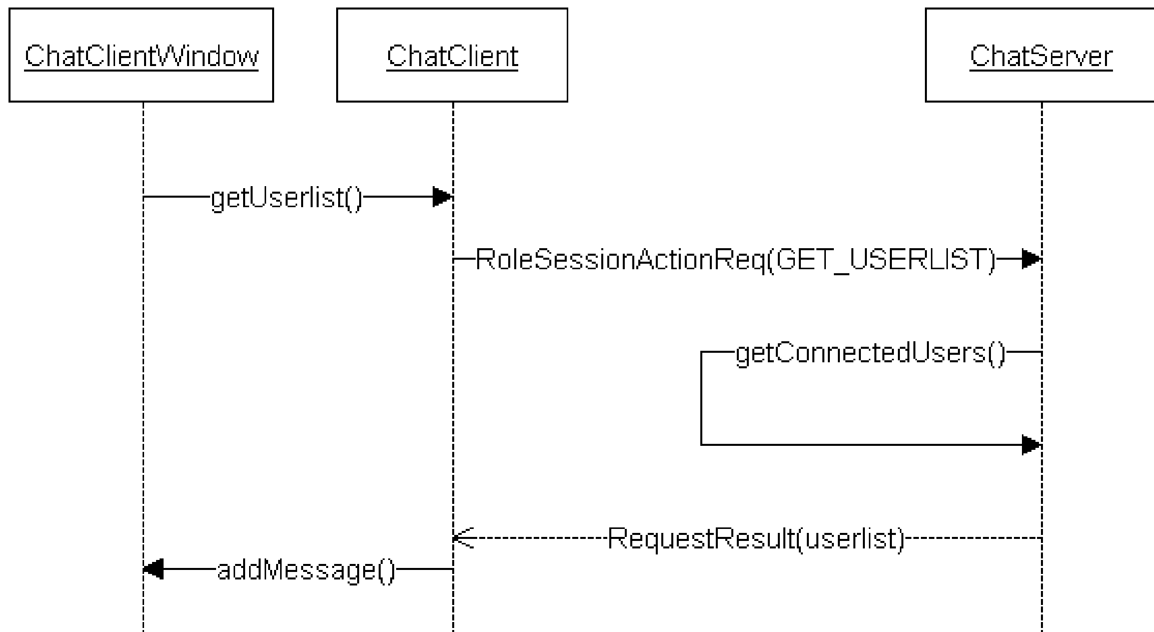

Figure 5.6: Sequence diagram for obtaining a list of all connected chatclients.

5.5.4 Send and receive message

Sequence diagram for the Send and receive message use case is shown in figure 5.7. A user can, through the chatclientwindow, write messages which when sent will be distributed to all the connected chatclients. In order to send a message the chatclientwindow calls the method sendMessage() in the chatclient. He adds the message to an applicationmessage of type CHAT_MESSAGE and sends it to the chatserver. The chatserver then forwards the applicationmessage to all the other chatclients. When the chatclient, who initially sent the message, receives notification that the message was successfully distributed, he adds the message to the log in his chatclientwindow.

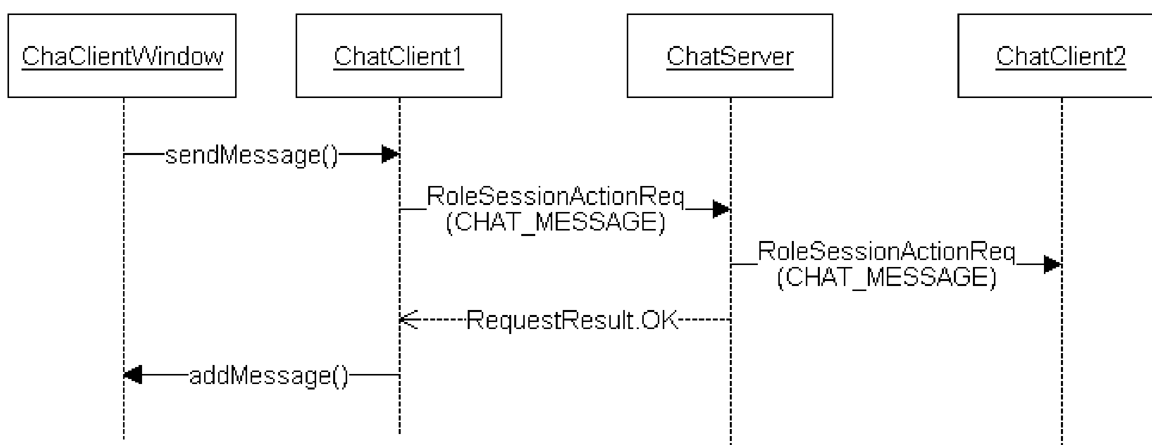

Figure 5.7: Sequence diagram for sending and receiving messages.

5.5.5 Suspend session

Sequence diagram for the Suspend session use case is shown in figure 5.8. A chatclient is told to suspend his session when he receives a SessionSuspend request from the useragent. If the chatclient is connected to a chatserver, he will try to disconnect from it. Then information about the chatclient's state is added to an ArrayList and returned to the useragent in the form of a RequestResult. Finally, the chatclient is plugged out by the director.

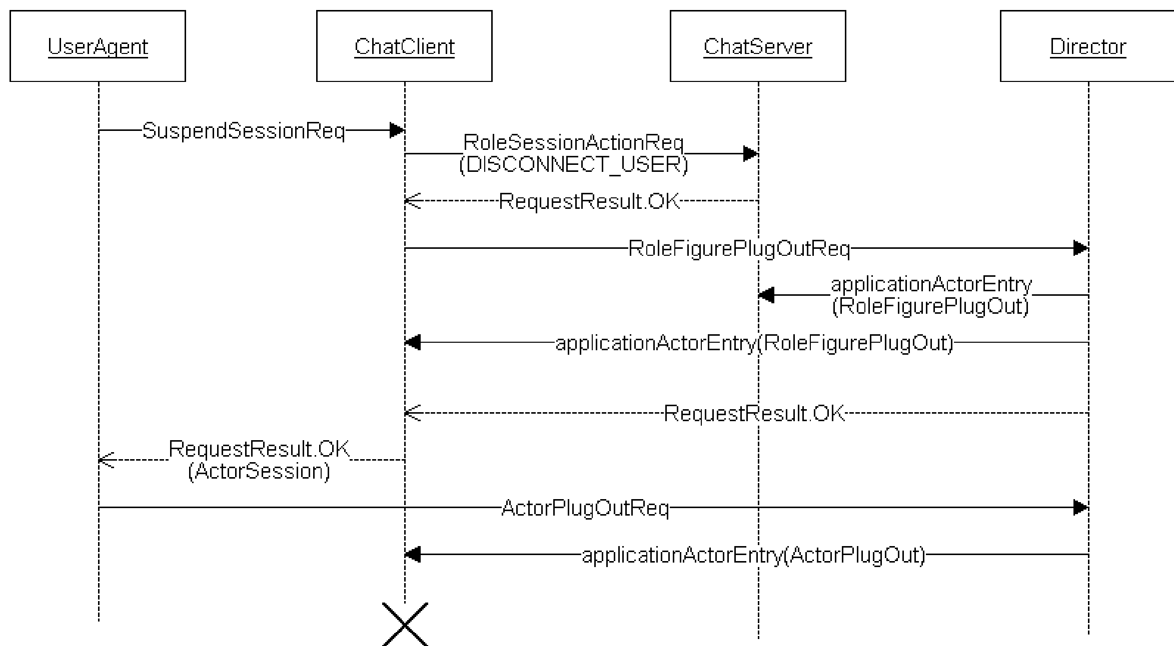

Figure 5.8: Sequence diagram for suspending a chatclient's session.

5.5.6 Resume session

Sequence diagram for the Resume session use case is shown in figure 5.9. It shows the sequences performed in order to resume a chatclients session from a stored sessiondescription. It is the useragent who is in charge of resuming the session. He first sends an ActorPlugIn request to the director who pluggs in a new chatclient on the user's node. If the chatclient was connected to a chatserver when the session was suspended, the useragent will try to restore this connection. He sends a RoleFigurePlugIn request to the director who creates a rolesession between the chatclient and a specified chatserver. If the chatserver no longer exists the connection can not be restored.

After the rolesession has eventually been created the UserAgent sends a SessionResume request to the chatclient. Added to the request is the list of variables which together forms the actor's state, like the username of the chatclient. If the connection to the chatserver is restored the chatclient needs to notify the chatserver that he wants to connect. This is done by sending an applicationmessage of type CONNECT_USER with his username to the chatserver.

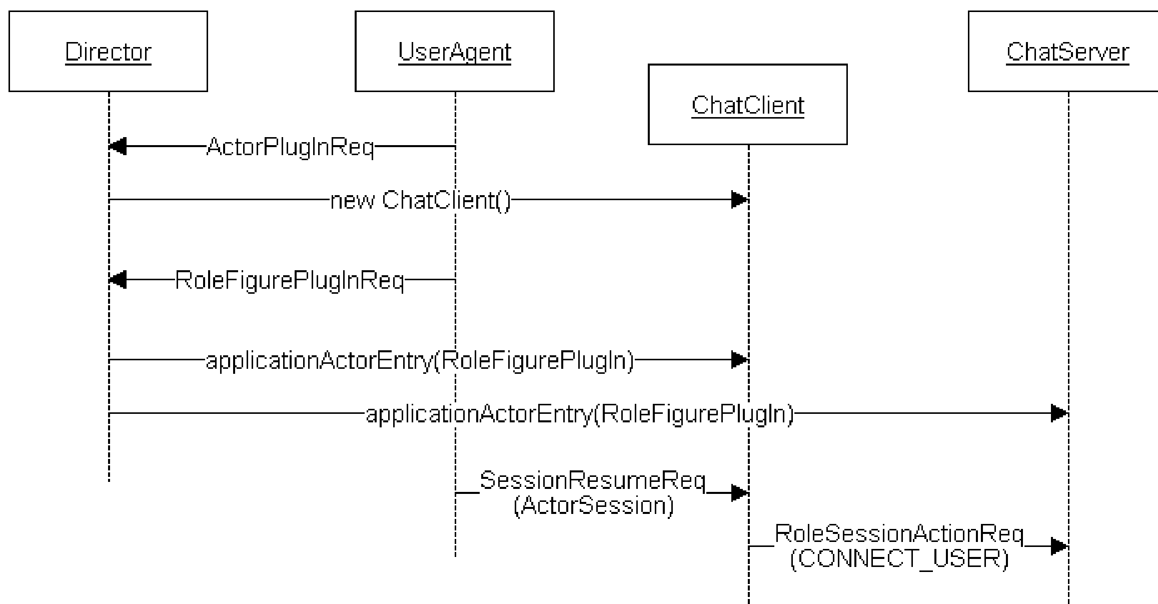

Figure 5.9: Sequence diagram for resuming a chatclient's session.

Chapter 6

The FileTransfer application

Contents

6.1 Requirements to the FileTransfer application 58
6.1.1 Functional requirements 58
6.1.2 Non-functional requirements 59
6.2 Use cases 60
6.3 Screenshot 61
6.4 Class diagram 62
6.5 Sequence diagrams 64
6.5.1 Send filetransfer request 64
6.5.2 Send filepart 65
6.5.3 Cancel transfer 66
6.5.4 Suspend transfer session 67
6.5.5 Resume transfer session 68

6.1 Requirements to the FileTransfer application

6.1.1 Functional requirements

The functional requirements for the FileTransfer application are stated below.

1. A ftclient (FileTransferClient) should be able to select a file on the user's node for transfer.

2. A ftclient should be able to send a filetransfer request to a specified ftclient giving him an option of either accepting or rejecting the file. If he accepts it the requesting ftclient should be able to transfer the file in parts of predefined size to the receiving ftclient.

3. Both the transmitting and the receiving ftclient should at any time be able to cancel the filetransfer.

4. A ftclient should be able to suspend his session if a SessionSuspend request is received. The minimum amount of information that should be stored is the name and path of the file selected to be transferred, and the Global Actor Identifier of the receiver of the file.

5. A ftclient should be able to resume a session that he has previously suspended if a SessionResume request is received. The information needed to resume the session must then be provided in the request.

6. The FileTransfer application should support multiple PaP domains, meaning that a ftclient in one domain should be able to transfer a file to a ftclient in another domain.

6.1.2 Non-functional requirements

The non-functional requirements for the FileTransfer application are stated below.

1. Usability - The FileTransfer application should be intuitive and easy to use.
2. Performance - The components that together form the application must not be so overwhelming and resource demanding that they become a bottleneck in the PaP system.
3. Space - Due to the fact that the components is going to be sent over the network the implementation should be as small and compact as possible to keep the transfer delay at a minimum.
4. Safety - The application must not include functionality that might be able to compromise any security of the network where the application is run.
5. Implementation - The functionality shall be implemented in 100% pure Java. The graphical user interface is to be built using Sun's Java Swing classes.
6. Design - The design must be as flexible as possible so that it can easily be extended with new functionality at a later time.

6.2 Use cases

The FileTransfer application consists of just one actor called FTClient (FileTransferClient). It can take on the role as a transmitter as well as a receiver. The way the application works is that a user sends a request asking if he can send a file to another user, instead of a user selecting and downloading a file from another user's machine. The user receiving the request can either accept the file or reject it.

In order to transfer a file one must go through two steps; send a filetransfer request and transfer the file. A ftclient can only transfer one file to one ftclient at a time. The file is divided into parts and transferred one part at a time. Both the receiver and the transmitter can at anytime cancel the current transfer. The application can also be suspended and later resumed again. Note that if a ftclient is participating in a filetransfer and is suspended, the filetransfer can only be resumed again if the suspending ftclient is the transmitter. This is due to complications with synchronization of the threads used in the application. The ftclient's state can be recreated in both cases, though.

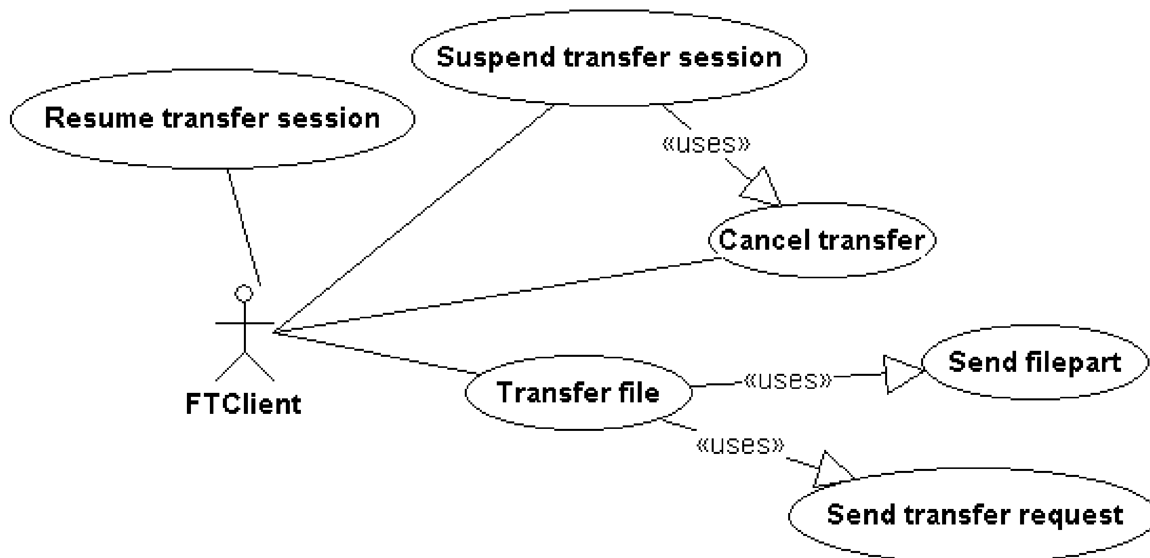

Figure 6.1: Use cases for the design of the FileTransfer application.

6.3 Screenshot

Figure 6.2 shows a screenshot of the FileTransfer application in action. As one can see FTClient2 is transferring the file RoleList.xml to FTClient1. When the screenshot was taken 40% of the file had been transferred. The graphical user interface is different for the two ftclients. This is because a ftclient contains two panels; a transfer panel and a receive panel. By default the transfer panel is showing but if the user accepts a file from another user the receive panel is shown.

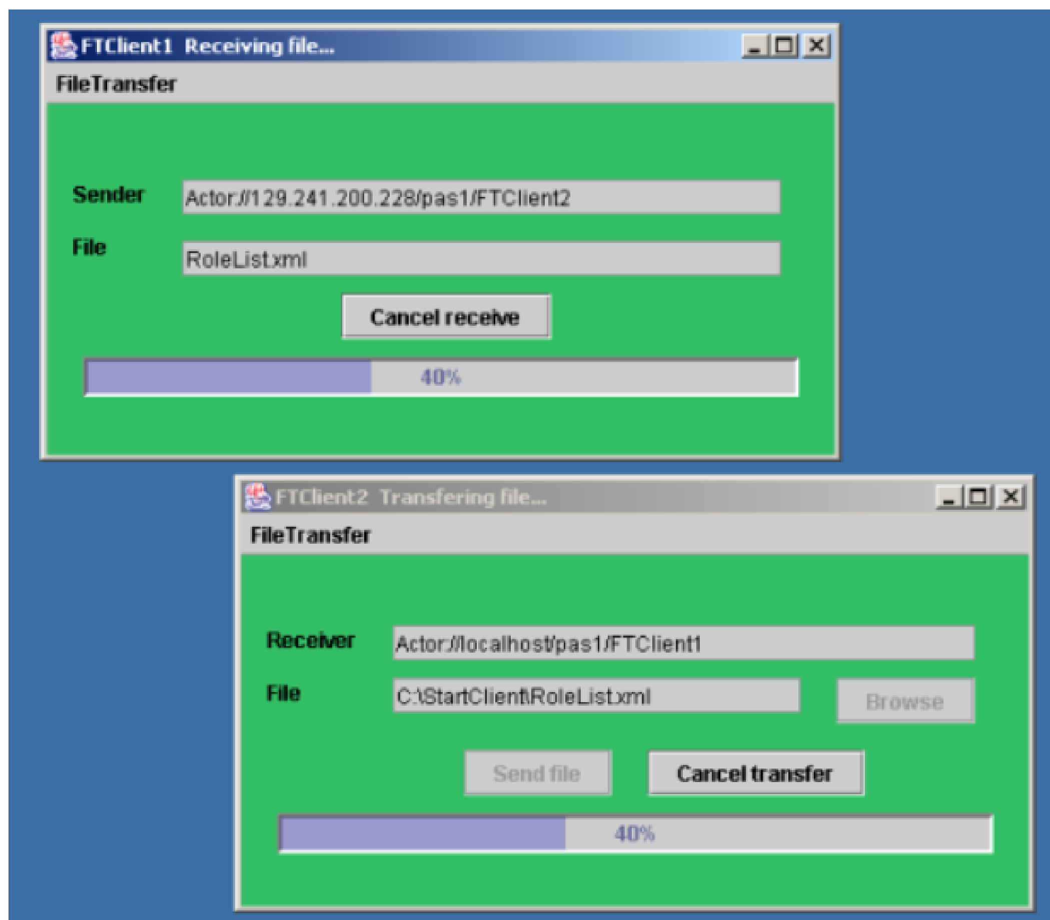

Figure 6.2: Screenshot of the FileTransfer application.

6.4 Class diagram

Figure 6.3 shows the classdiagram of the FileTransfer application. As mentioned earlier the application consists of only one actor called ftclient. The ftclientWindow is the graphical user interface of the application. It is in charge of enabling and disabling the GUI components depending on whether the ftclient is transferring or receiving a file. Transferthread is a thread that is started when a filetransfer is either started or resumed. It is responsible for reading the file to be sent, dividing it into parts of size equal to the bytesize property and sending the parts to the other ftclient. The thread can be stopped by setting its finished property to true.

The FTClient class contains a number of constants defining the different types of application-messages. They are used when a message is sent as a RoleSessionAction request. The ArrayList called fileinfo contains information about the current filetransfer. This information is used if the ftclient is suspended and it is included with the rest of the actor's state information.

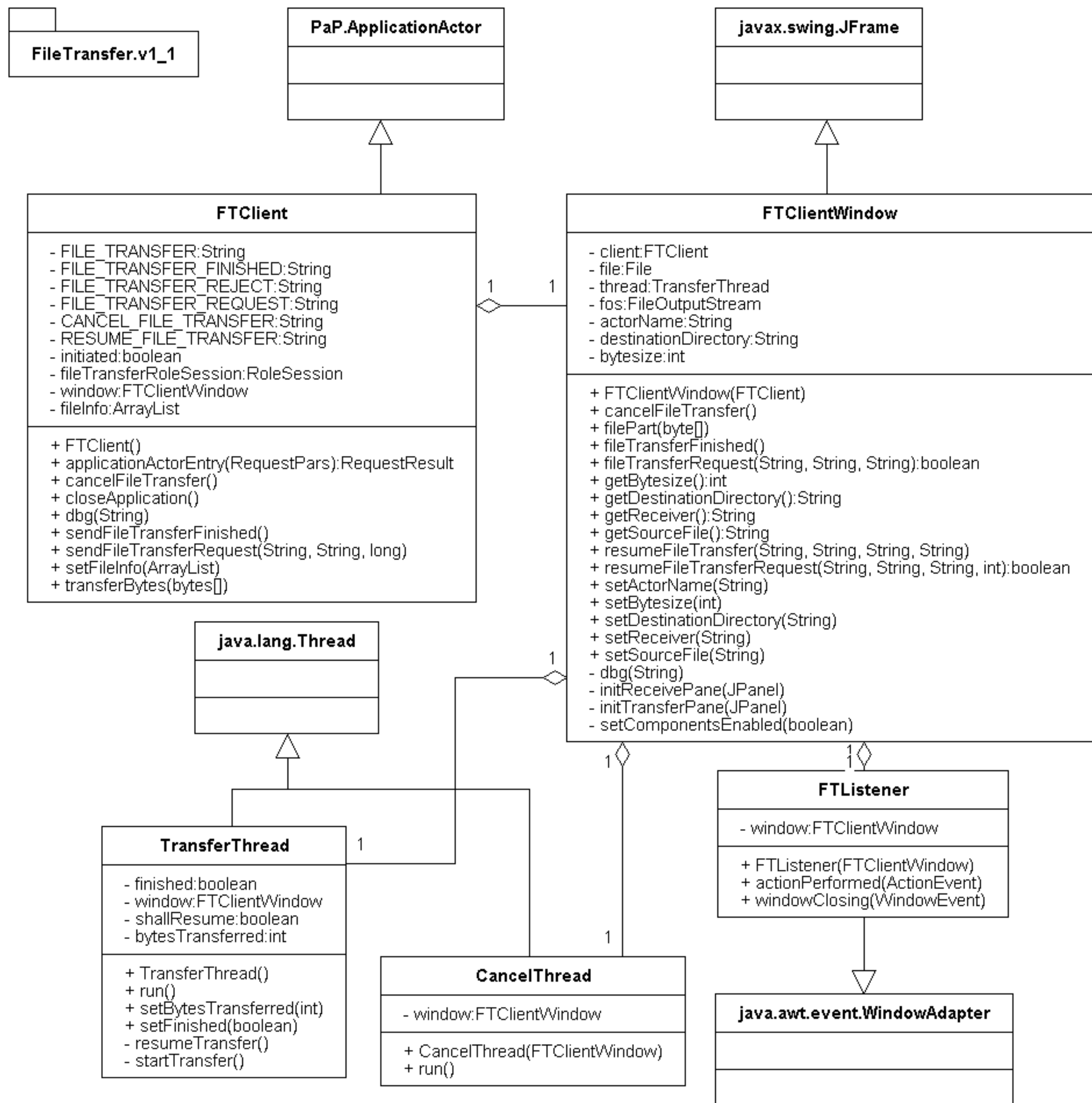

Figure 6.3: Class diagram for the FileTransfer application.

6.5 Sequence diagrams

6.5.1 Send filetransfer request

Sequence diagram for the Send transfer request use case is shown in figure 6.4. When a user wants to transfer a file he provides the GAI of the receiving ftclient, selects a file on his node and presses the [Send file] button. The ftclient will then start a new transferthread. The thread will create a rolesession between the two actors by sending a RoleFigurePlugIn request to the director. Then an applicationmessage of type FILE_TRANSFER_REQUEST is sent to the other actor asking him if he accepts the file. If he accepts it the tread will start to transfer the file as shown in 6.5.2.

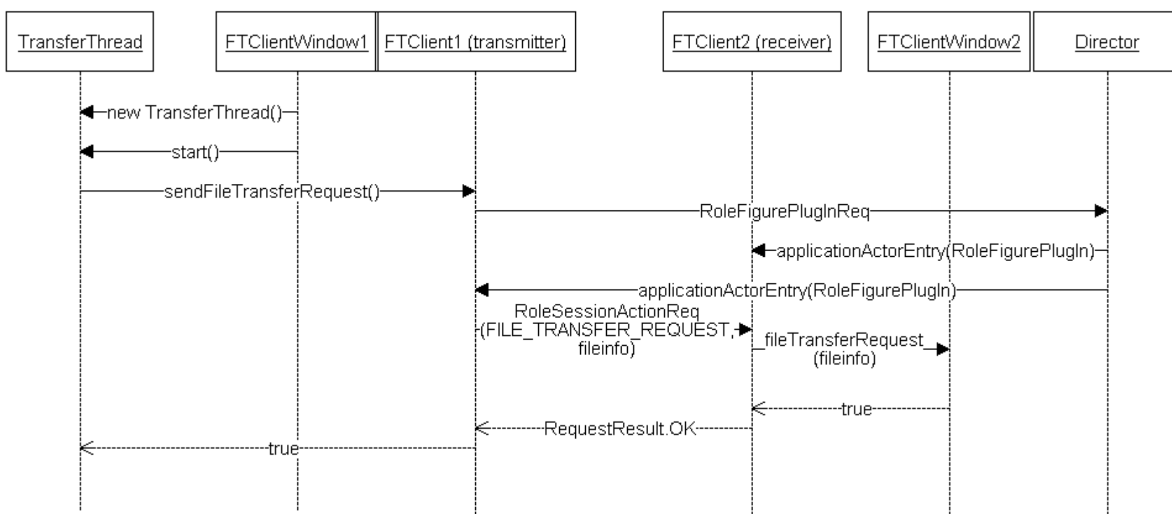

Figure 6.4: Sequence diagram for sending a filetransfer request.

6.5.2 Send filepart

Sequence diagram for the Send filepart use case is shown in figure 6.5. It assumes that the rolesession between two ftclients have been created and that the receiving ftclient has accepted the file. The transferthread opens a FileInputStream to the file, reads parts of the file, adds the content to a byte array and sends it to the other ftclient in an application-message of type FILE_TRANSFER. This sequence of operations continues until the thread is stopped or the whole file has been transferred. If the latter is the case an application-message of type FILE_TRANSFER_FINISHED is sent to the other ftclient to notify him. Either way the rolesession is removed by sending a RoleFigurePlugOut request to the director.

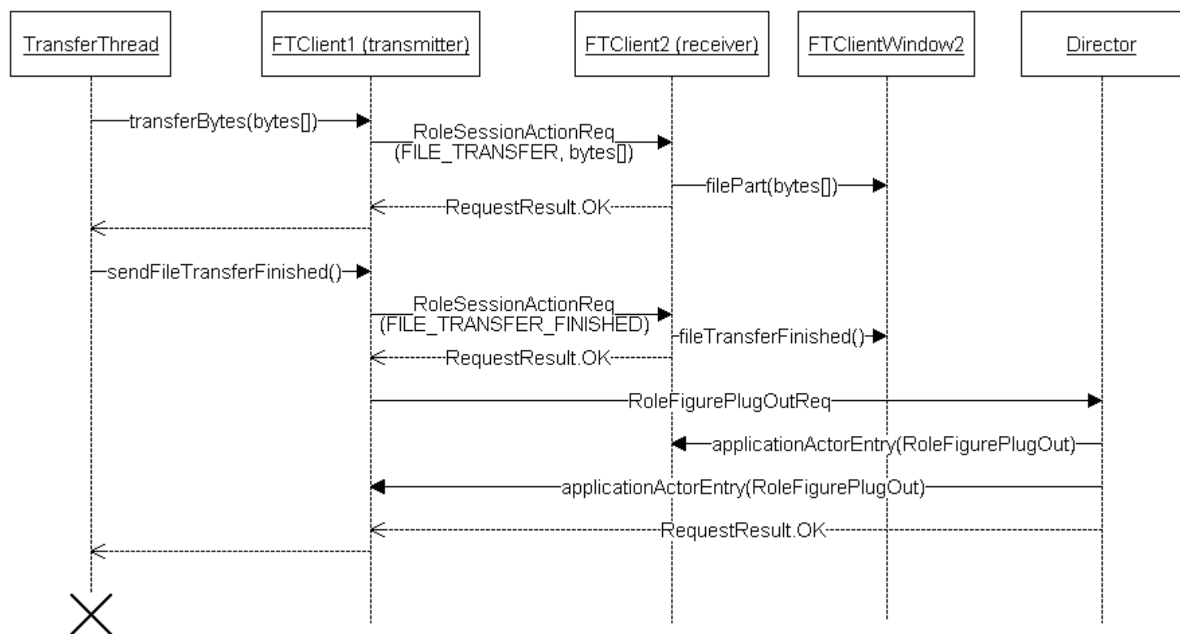

Figure 6.5: Sequence diagram for transfering a file.

6.5.3 Cancel transfer

Sequence diagram for the Cancel transfer use case is shown in figure 6.6. Both the transfering and the receiving ftclient can anytime cancel the filetransfer. The figure shows what happens when the transfering ftclient cancels the filetransfer. A similar sequence of operations is performed if the receiving ftclient cancels the filetransfer.

When the user presses the button [Cancel tranfer] the ftclientWindow sets the finished property in the transferthread to true. This property is checked before each new filepart is sent, and setting it to true will make the tread stop the transfer. Then the ftclientWindow will call the method cancelFileTransfer() in the ftclient which will send an application-message of type CANCEL_FILE_TRANSFER to the receiver to notify him. Finally, the rolesession is removed by sending a RoleFigurePlugOut request to the director before the GUI components is reset again.

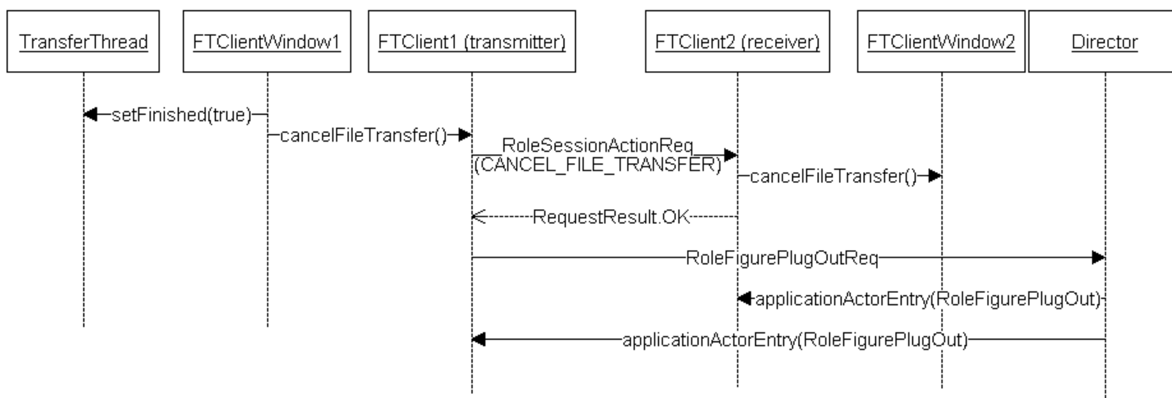

Figure 6.6: Sequence diagram for cancelling a filetransfer.

6.5.4 Suspend transfer session

Sequence diagram for the Suspend transfer session use case is shown in figure 6.7. A ftclient's session is suspended when the useragent sends a SuspendSession request to him. If the actor is not engaged in a filetransfer, information about its state is returned to the useragent. If it is currently engaged in a filetransfer the filetransfer is canceled as specified in 6.5.3. Then the information in the ArrayList fileinfo is added to the actor's state information and returned to the useragent. In both cases the useragent sends an ActorPlugOut request to the director and the ftclient is plugged out.

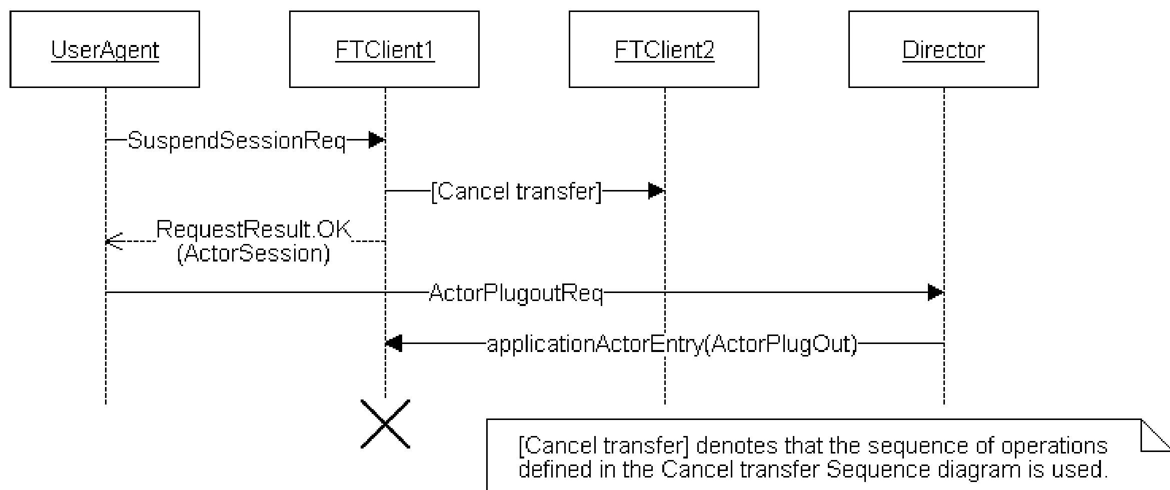

Figure 6.7: Sequence diagram for suspending a ftclient's session.

6.5.5 Resume transfer session

Sequence diagram for the Resume transfer session use case is shown in figure 6.8. Due to the design of the FileTransfer application a suspended filetransfer can only be automatically resumed if it is the tranfering ftclient that was suspended. The figure shows what happens if this is the case. A new ftclient is plugged in and the rolesession to the receiving ftclient is recreated. If the receiving ftclient is unavailable the rolesession can not be recreated. The actor's session is resumed when the useragent sends him a SessionResume request, with the actor's state information added.

If the rolesession was successfully created an applicationmessage of type RESUME_FILE_TRANSFER is sent to the other ftclient asking if he wants to resume the suspended filetransfer. If he accepts, a new transferthread is created and configured to resume the filetransfer. The thread will then start sending fileparts as if nothing had happended.

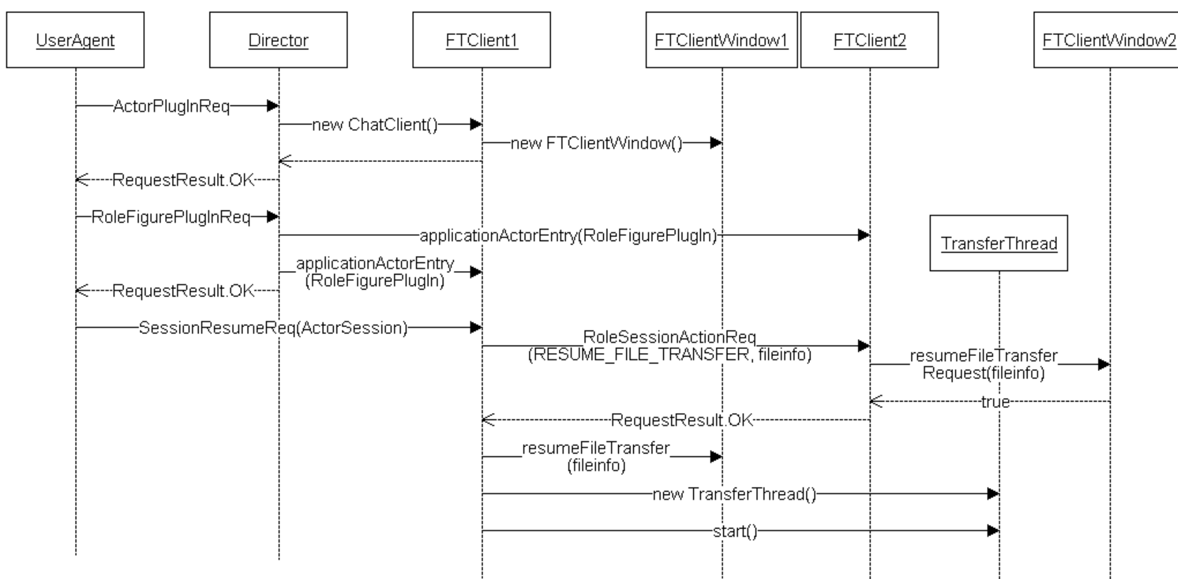

Figure 6.8: Sequence diagram for resuming a ftclient's session.

Chapter 7

The results

Contents

7.1 Problems encountered **70**

7.1.1 Plugging out actors 70

7.1.2 Discrimination of certain types of requests 70

7.1.3 Deadlock of threads 71

7.1.4 Introducing new types of requests 71

7.1.5 Start-up of the PaP system 72

7.2 Testing **73**

7.3 Future improvements **74**

7.1 Problems encountered

As always in larger projects some problems were encountered in this thesis. Some of them were extremely intricate to discover and solve. This chapter will present the most serious ones for two reasons. The first reason is to make other people aware of them if they encounter the same or similar problems in the future. The second reason is to outline what can be improved with the PaP system if a redesign or reimplementation is to take place in the future. This could be done as a project, a thesis or perhaps as a summerjob. Note that not all of the problems presented are problems with the PaP system but are general problems encountered.

7.1.1 Plugging out actors

According to the PaP architecture an actor can not be plugged out when he participates in more than one rolesession. This means that the actor is either the initiator or the cooperator of the rolesession. When the user is logged out from his home domain his useragent sends an ActorPlugIn request to the director requesting to plug-in a new loginagent on the user's node. The useragent then sends an ActorPlugOut request to the director requesting to plug-out himself. However, now the useragent participates in both his initial rolesession and the loginagent's initial rolesession, and the useragent is unable to be plugged out.

The problem was solved by commenting out the code checking the number of rolesessions an actor was participating in and in certain places hard-code the value of some variables. This is a dreadful but effective solution, but should be solved in a more appropriate way in the future. This might call for a redesign of part of the PaP system, though, since it affects some core functionality.

7.1.2 Discrimination of certain types of requests

One of the features of the Chat application is that a chatclient can obtain a list from the chatserver of all the currently connected chatclients. This is done by sending a RoleSession-Action request with an applicationmessage of type GET_USERLIST to the chatserver. The chatserver then generates the list and returns a requestresult with the list added to it. This functionality did not work since the returned requestresult was emtpy, although the one sent was not.

The problem was a feature implemented in the PAS. Almost all requests received by the PAS is forwarded to either the PNES or to the specified actor. The exception is requests of type RoleSessionAction and SubscribeReport. They are specifically filtered out and added to a vector. The PAS starts a separate thread in charge of checking the vector and forwarding the requests it contains. The result was that the returned requestresult was interchanged with an empty one. The reason for this feature was to support asynchronous messages between different PAS instances, have a looser coupling between them and to try to prevent deadlocks. The intention was good but it made it possible to implement the described functionality.

The quick solution was to only filter out SubscribeReport requests in the PAS, but a better solution to the problem is yet to be discovered. However, to perform a synchronous operation with asynchronous messages is not possible. A property in the class RequestPars, specifying whether the request is to be sent synchronous or asynchronous, might be a better solution. That way all types of requests can be sent asynchronous but then it is up to the sender to decide.

7.1.3 Deadlock of threads

Part of the Jave language is a GUI framework called Swing. When a user interacts with a GUI component, like a button, an event is generated, in this case an ActionEvent. In order to perform an operation when a button is pushed, one would have to add a so called ActionListener to the button. The lister implements the ActionListener interface and when the button is pushed the method actionPerformed() is called. This is the way a listener is notified of ActionEvents. The EventDispathThread is responsible for both drawing the GUI components and delegating events to the event listeners.

Java is a synchronous language so when a method is called it has to return before the program flow can continue. When a user chooses to logoff from a domain he selects the 'Log off' menuitem in the userwindow. This generates an ActionEvent and the EventDispatchThread calls the method actionPerformed() in the class UserWindow. The EventDispatchThread is now paused until the method returns. The userwindow calls the method closeApplication() in the class UserAgent. The useragent sends an ActorPlugOut request to the director requesting him to plug-out himself. The director will then send an ActorPlugOut request to the user's PAS who notifies the actor that he is being plugged out by calling the method applicationActorEntry() in the class UserAgent.

A normal operation would now be to call the method dispose() in the class UserWindow, to remove the GUI. However, this method-call is put on the so called event-stack that the EventDispatchThread is responsible of handling. Since this thread is waiting for the call to the ActionListener in the class UserWindow to return a deadlock has occured.

The solution was to start a new thread when a useragent is plugged out, which only removes the GUI. Deadlocks can easily happen when you develop applications in a multithreaded system. The ftclient actor uses several threads to perform different operations such as transfer of a file and show a popupbox. Since the author was unaware that this situation could happen, the design of the FileTransfer application is a little bit flawed and can benefit from a redesign in the future. Unfortunately, a very limited timetable did not allow for such a redesign.

7.1.4 Introducing new types of requests

The mobility framework introduces seven new types of requests. They are all defined in the class RequestPars. The first time one of the new requests was sent from the director to an actor it did not come through, but was rerouted back to the director. I first thought that the actor's GAI was misspelled but it was correct. After quite some debugging the cause of the problem was found in the class ApplicationActor. This is the superclass of

all actors except the director actor. A request is forwarded from the PAS to an actor by calling its method actorEntry(). There the type of request is checked, appropriate operations are performed and the actor is notified about the request by calling the method applicationActorEntry() that all actors inheriting ApplicationActor must implement. All requests of an undefined type are sent back to the director in order to be served by him. This feature can either be seen as a bug or as part of the design. A better solution would probably be to return a requestresult of type FAIL instead of forwarding it to the director, since it would signal that the actor was unable to serve the request. The solution was to check for the new types of requests and forward them to the actor.

7.1.5 Start-up of the PaP system

One of the biggest problems with the PaP plattform is the start-up of the system. It takes forever from you send the first PlayPlugIn request until the director is up and running and the play is plugged in. The cause of the problem was first suspected to be the downloading of class-files from the webserver. A subproject was started to reimplement part of the PaP system to enable downloading of class-files from a local directory instead of downloading them from a webserver. The project was terminated after a week because such a feature demanded too many changes in the PaP system. The feature is very useful, though, and should be incorporated in the future.

Later the cause of the problem was discovered. The problem is the start-up of a new PAS on a node. Or more accurately, the start-up of a new PAS on a node running the Microsoft Windows operating system. When the PaP system is run on a machine running the Linux operating system, it takes only 6 or 7 seconds from the first PlayPlugIn request is sent until the director is up and running and the play is plugged in. When run on Windows the start-up can takes a minute or even longer. The reason for this is still a mystery, but until the problem is discovered and solved it is clearly recommended that future development, testing and debugging is to be performed on a computer running Linux. This would tremendously decrease time used on testing and boost development.

The testcases listed in the appendix were first performed on a computer running Linux. Everything works fine when both the director and the actors are running on the same node or computer. However, when the director was put on a separate computer there were some problems with the URLs. They were all set to be localhost and the communication between the nodes failed. The problem was either a bug in the Java implementation used or the computers were some way misconfigured. The problem needs to be looked into in the future.

7.2 Testing

A set of test-cases are listed in appendix A. They are test-cases for testing user and session mobility within one domain and between different domains. All the test-cases performed correctly. In addition the mobility framework has been thoroughly tested, which includes the use of the three agents (LoginAgent, VisitorAgent and UserAgent) and the XMLFileUtil class. One can therefore conclude that the framework fulfil all the requirements in section 3.1.

The two example applications, Chat and FileTransfer, have also been thoroughly tested. The session of a chatclient can be completely resumed so that the connection to the chat-server is restored, the username is set correctly and the chatclient is connected. The session of a ftclient can also be completely resumed so that the ftclient automatically resumes a suspended filetransfer. Partially due to the problem mentioned in 7.1.3 a filetransfer can only be resumed if the transmitting ftclient is suspended. However, the application is designed so that a filetransfer is supposed to be initated by the transmitter, so lack of this feature might not be so bad after all. Both the applications visually demonstrate the capabilities of the mobility framework as was intended.

7.3 Future improvements

In section 7.1 several problems with the PaP system were described. In addition to solve those problems improvements can be made to both the mobility framework and the example applications. Some of them are listed below.

- Adding support for actor and terminal mobility. As mentioned earlier this is planned sometime in the future.

- Have visitoragents read an XML-file with a list of GAIs of directors other than their current director. This is useful when users want to logon to their home domain from another domain. Then they will be able to select the GAI of their home director from the list and do not have to provide the exact GAI manually.

- Implement some kind of search and discover functionality of actors. This would be useful in the Chat application so that one does not have to know the GAI of the chatservers, but could instead search for chatservers by their names. The feature could also be used in the situation mentioned in the previous entry.

- Make the useragent more resilient to failure when a user's session is resumed. By plugging in all the actors first and then create their rolesessions, the chance of succeeding increases. This relates to the situation when a rolesession is to be created between two actors listed in the user's sessiondescription. Normally only one of them has stored information about the rolesession, and the rolesession is re-created when that actor is re-created. If two actors are not created a rolesession between them can obviously not be created. Therefore the order they are re-created decides whether the rolesession can be re-created or not.

- Reimplement the FileTransfer application so that both the transmitter and the receiver can initiate resuming a suspended filetransfer. As mentioned earlier the current version only enables the transmitter to do this.

Conclusion

The objective of this thesis was to create a partial framework for mobility in the PaP architecture by implementing support for user and session mobility in the PaP platform. The framework is intended to support four generic categories of mobility; user, session, actor and terminal mobility. Support for actor and terminal mobility is planned added in the near future.

A number of test-cases have been performed and they verify that the new functionality is able to provide user and session mobility within one domain as well as between different domains. Three agents are used by the framework and they all provide a simple and intuitive graphical user interfaces to the PaP system. The visitoragent and the useragent enables a user to plug-in and plug-out actors by following a simple two-click procedure. Normally this is performed by manually writing commands through a PNES' baseframe window. The two agents therefore increases the user-friendliness of the PaP system. Another convenient feature introduced is the possibility for a user to customize the size, location and background color of his useragent. These properties are stored in a user's userprofile and a user therefore only needs to set them once.

A functionality lacking in the PaP system was the possibility to create rolesessions between two PaP entities by a third entity. This functionality was needed in order to enable a useragent to re-create an actor's rolesessions when a user's session was resumed. Recreation of an actor's rolesessions is a vital part in this process. Two new types of requests were therefore defined. They are called RoleFigurePlugIn and RoleFigurePlugOut which creates a new and removes an existing rolesession between two PaP entities, respectively. This type of functionality also enables more complex applications, such as the useragent, to be created.

Two examle applications were implemented to demonstrate the capabilities of the mobility framework. They utilize the new functionality, and a chatclient can therefore be automatically reconnected to a chatserver when its session is resumed. Similarly, a suspended filetransfer can automatically be resumed when a ftclient's session is resumed.

Several problems were encountered in this thesis. Some of them were caused by design flaws in the PaP system. All the problems were solved, but in order for user and session mobility to work, it required some changes to the PaP system that were less fortunate. They should be solved in a more appropriate manner in the future if a reimplementation of the whole or part of the PaP system is to take place.

Bibliography

[1] Meling, H. *Plug'n'Play for Networks and Teleservices* [online]. Available from: http://www.item.ntnu.no/\~plugandplay/ [Accessed 6 May 2002]

[2] Aagesen, F. A., Helvik, B. E., Wuwongse, V., Meling, H., Bræk, R., Johansen, U. *Towards a plug and play architecture for telecommunications* [online]. Paper presented at the IFIP TC6 Sixth International Conference on Intelligence in Networks, Vienna, November 1999. Available from: http://www.item.ntnu.no/\~plugandplay/publications/smartnet.pdf [Accessed 6 May 2001].

[3] Aagesen, F. A. *Plug and Play (PaP) for Telecommunications - Architecture and Demonstration Issues* [online]. Presentation presented at The International Conference on Information Technology for the new millennium (IConIT), Bangkok, Thailand, May 2001. Available from: http://www.item.ntnu.no/\~plugandplay/publications/IConIT.pdf [Accessed 6 May 2002].

[4] Aagesen, F. A., Helvik, B. E., Johansen, U., Meling, H. *Plug and Play for Telecommunication Functionality - Architecture and Demonstration Issues* [online]. Paper presented at The International Conference on Information Technology for the new millennium (IConIT), Bangkok, Thailand, May 2001. Available from: http://www.item.ntnu.no/\~plugandplay/presentations/IConIT-2001.pdf [Accessed 6 May 2002].

[5] Meling, H. *PaP Demonstrator* [online]. Available from: http://www.item.ntnu.no/~plugandplay/PaPdemo.shtml [Accessed 23 May 2002]

[6] Meling, H. [online]. Available from: http://www.item.ntnu.no/~plugandplay/documentation/SystemDoc/Main/Main.pdf [Accessed 23 May 2002]

[7] Malek, M., Aagesen, F. A. *Mobility management in Plug and Play network architecture*. Department of Telematics, Norwegian University of Science and Technology, Trondheim, Norway. Paper presented at IFIP TC6 Seventh International Conference on Intelligence in Networks, Saariselka, Finland, April 2002.

[8] Mazen Shiaa, M., Liljebäck, L. E. *User and Session Mobility in a Plug-and-Play Architecture*. Paper presented at IFIP WG6.7 Workshop and EUNICE Summer School, Trondheim, Norway, September 2002.

[9] *Extensible Markup Language (XML)* [online]. Available from: http://www.w3.org/XML/ [Accessed 6 May 2002].

Appendix A - Test-cases

This appendix contains a set of test-cases performed on the PaP system. Each test-case describes a scenario of what kind of operations are being performed, as well as describing the results of the operations. The first section lists test-cases for testing user mobility. The second section lists test-cases for testing session mobility. The actor used is a chatclient but another type of actor could also be used. Note that the chatclient does not connect to a chatserver. This could be done to test that the actor's rolesessions are re-created when a user's session is resumed.

Testing user mobility

1. Scenario: Logon to local director. Do not select to resume your session. Change the useragent's settings and suspend your session. Logon to local director again and this time select to resume your session.

   Result: The settings are stored in your userprofile and set correctly when you are logged on.

2. Scenario: Logon to local director. Do not select to resume your session. Change the useragent's settings and suspend your session. Logon to local director again on another computer that has the same director and this time select to resume your session.

   Result: The settings are stored in your userprofile and set correctly when you are logged onto the new computer.

3. Scenario: Logon to local director. Do not select to resume your session. Change the useragent's settings and suspend your session. Logon as a visitor on a computer with a different director. Logon remotely from the visitoragent to your home domain and this time select to resume your session.

   Result: When you logoff the settings are stored in your userprofile. When you logon a new PAS is created on the user's node. This PAS has a homeinterface equal to the local director. The useragent is plugged in, the user's session is resumed and the settings are set correctly.

4. Scenario: Continue where the previous test-case left off. You are logged on to your home domain from another domain. Change the useragent's settings and suspend your session. Logon to local director again on the first computer used and select to resume your session.

   Result: The settings are stored in your userprofile and set correctly when you are logged on.

Testing session mobility

1. Scenario: Logon to local director. Do not select to resume your session. Plug-in a chatclient actor. Change the username used by the chatclient and suspend your session. Logon to local director again and this time select to resume your session.

   Result: After the useragent is plugged in your session is resumed. The chatclient is plugged in and the username it set correctly.

2. Scenario: Logon to local director. Do not select to resume your session. Plug-in a chatclient actor. Change the username used by the chatclient and suspend your session. Logon to local director again on another computer that has the same director and select to resume your session.

   Result: After the useragent is plugged in your session is resumed. The chatclient is plugged in and the username it set correctly.

3. Scenario: Logon to local director. Do not select to resume your session. Plug-in a chatclient actor. Change the username used by the chatclient and suspend your session. Logon as a visitor on a computer with a different director. Logon remotely from the visitoragent to your home domain and select to resume your sesson.

   Result: After the useragent is plugged in your session is resumed. The chatclient is plugged in and the username is set correctly.

4. Scenario: Continue where the previous testcase left off. You are logged on to your home domain from another domain. Change the username used by the chatclient and suspend your session. Logon to local director again on the first computer used and select to resume your session.

   Result: After the useragent is plugged in your session is resumed. The chatclient is plugged in and the username is set correctly.