MOBILITY MANAGEMENT IN PLUG AND PLAY NETWORK ARCHITECTURE

Mazen Malek, Finn Arve Aagesen
Department of Telematics
Norwegian University of Science and Technology
Trondheim, Norway
malek@item.ntnu.no , aagesen@item.ntnu.no

Keywords: Plug and Play, mobility management, theatre model, software agents.

Abstract:

A Mobility Management platform in Plug-and-Play (PaP) network architecture is presented. Four different approaches for ***Actor***, ***Terminal***, ***User*** and ***Session*** Mobility Management are illustrated. We explore a few issues related to implementation design and propose a set of components to facilitate the deployment of this platform in the available PaP applications. The PaP architecture is briefly introduced and subsequent definitions and terms are dealt with. The architecture itself is based on a theatre metaphor, in which 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. The mobility management is introduced to add a capability to handle any move of any constituent of the system. This could range from moving code agents, or actors, to moving users of the system. Throughout the paper we try to give a survey on the various mobility cases and make an effort to demonstrate an early set of mobility management algorithms or methods.

1. INTRODUCTION

Grade of network intelligence is defined as *the efficient flexibility* in the introduction of new teleservices and the *efficient flexibility* in the execution of teleservices. IN (Intelligent Networks) [ITU92], TINA (Telecommunication Information Networking Architecture) [TINA95], Mobile Agents and Active Networks ([Bies97], [Bies98], [Raza99], [Tenn97]) are all solutions aimed to improve the network intelligence.

Plug-and-play (PaP) for telecommunications means that the hardware and software parts 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. Plug-and-play functionality means utterly increase of network intelligence. The concept PaP stems from the personal computing area. PaP simply means that you plug-in and then the system works. In these systems, the plugged in component as well as the framework has a *predefined* functionality. We denote this *static* PaP. A more general kind of PaP is when the plugged-in unit has a set of basic capabilities, but its functionality is defined as a part of the plug-in procedure and it can be changed dynamically. We denote this as *dynamic* PaP. An example is a cellular phone, which obtains the services it provides depending on its inherent capabilities, which user that logs on, and which network it is attached to. *The focus of this paper is on dynamic PaP, and* from now on *PaP* means *dynamic* PaP. For a detailed description of the PaP architecture and the ongoing research activities around this project please refer to: (http://www.item.ntnu.no/~plugandplay),

For purposes of more flexible computational capabilities, utilisation and adaptation of wider range of network services and possible introduction of network management tasks, our PaP model should be enhanced with various mobility management schemes. The very basic requirement in this regard is the ability to move functionality among nodes or components. In this paper we refer to this by *Actor Mobility*. Actor is the core concept of the PaP architecture and it behaves according to a prescribed role or behaviour. *Actor Mobility* has little to do with mobile agent technology [Bies97] though they share the same vision of autonomous computational entities moving around the network. In our case the actors are just a representation of the functionality at a given node. When actors move they just transfer their functionality, or by other words move the execution of the functionality to a different node. Section 3 provides a clearer view on this concept. Nowadays telecommunication networks provide the users with ever improving and flexible mobility in both wired and wireless technologies. Users could move their terminals more and more freely, and yet be able to access the same range of services. This ability adds a great challenge in terms of managing terminal mobility. So, as the PaP architecture means for an intelligent system that is applicable for any environment it should be no exception. *Terminal Mobility* is handled in section 4. For the remaining types of mobility, the *User and Session mobility*, sections 5 and 6 give an overview of the proposed methods. Finally we give *summary and conclusions that include our further interests and future work.*

2. A PaP Reference Architecture

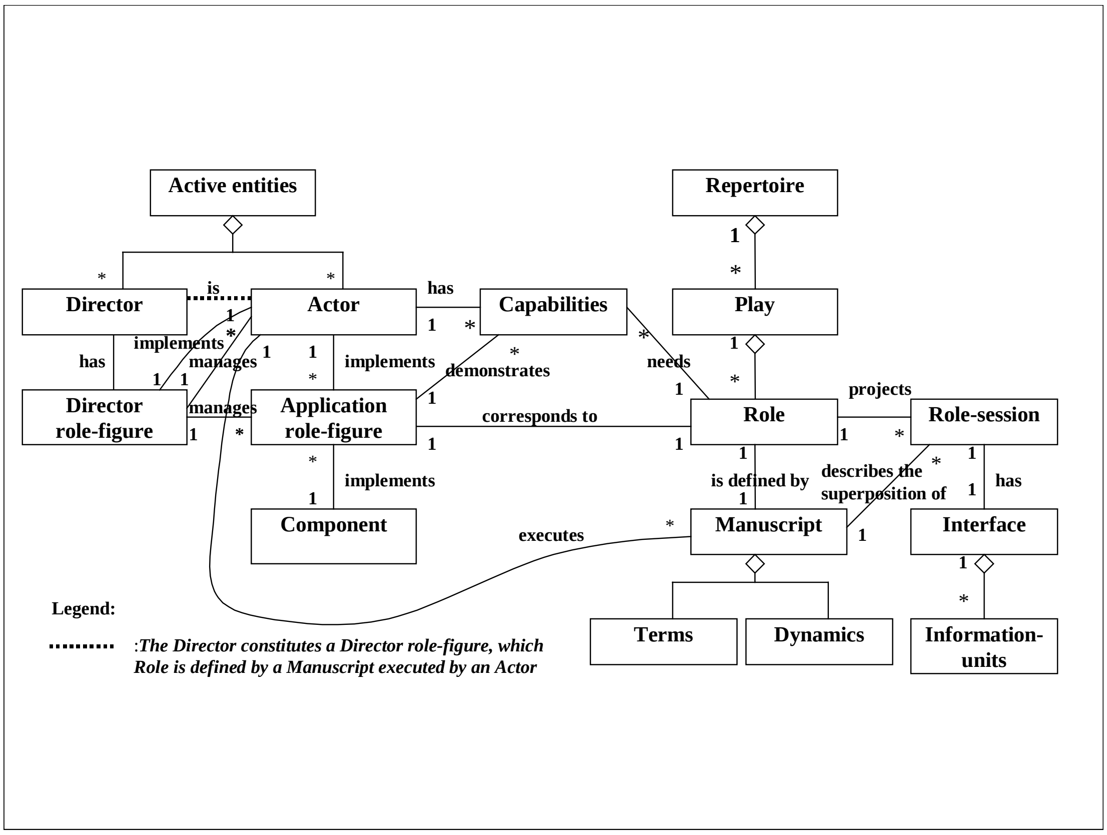
In this section we give a brief description of the PaP architecture, how does it define functionality and what are its basic constituents. The PaP was introduced in [Aage99]. Important definitions are highlighted and the basic procedures are mentioned. Some figures illustrating the different object models and view points are also included to add clarity.

PaP components: are real-world active hardware and/or software modules. These can be *combined hardware/software* modules with one or more external hardware interfaces, or *pure software modules*. These must interface with a software platform capable of running PaP application software.
The functional object model: PaP components are composed from (one or more) interacting instances of PaP functional objects, where each instance is defined by reference to an object type. This means that the PaP *component* functionality is defined by a *functional object model* consisting of *functional PaP objects*. ISO's reference model for Open Distributed Processing (ODP) [Duts96] defines the enterprise, computational, information, engineering and technical viewpoints. The viewpoints of *primary interest* with respect to PaP are the *computational* and the *engineering* viewpoints. The PaP components are basically engineering viewpoint objects. The relationship between important PaP concepts is shown in Figure 1.
PaP Actor: An actor is a generic object with a generic behaviour. Actors are able to behave according to a *manuscript*. The *repertoire* consists of *plays*, which are defined by *roles* and the role is formalised by a *manuscript*. These concepts have meaning similar to as those that

are used in the theatre context. The manuscript is the functional PaP object type definition. An instance of a PaP functional object, also here denoted as a *role-figure* is realised by an actor, which is executing the manuscript.

PaP Manuscript: defines the *entire* behaviour of an actor. R*ole-session* is a projection of the behaviour of the actor with respect to one of its interacting actors. The entire role as well as the role-sessions are EFSMs (Extended Finite State Machines). An actor also has a defined set of *capabilities*, which is the ability or power to do something. The capabilities are the result of the available hardware functionality connected to the hardware executing the actor software behaviour, but also the quantitative aspects such as processing capacity [Aage99].

PaP Play: is a defined autonomous functionality. The play defines the context for relationships between PaP objects as well as their behaviour. One important PaP object instance necessary to initialise any play is the *director role-function*. Director behaviour is also defined by an instance of a play. An actor has three distinct behaviour phases: *1)* the plug-in phase, *2)* the play phase and *3)* the plug-out phase.

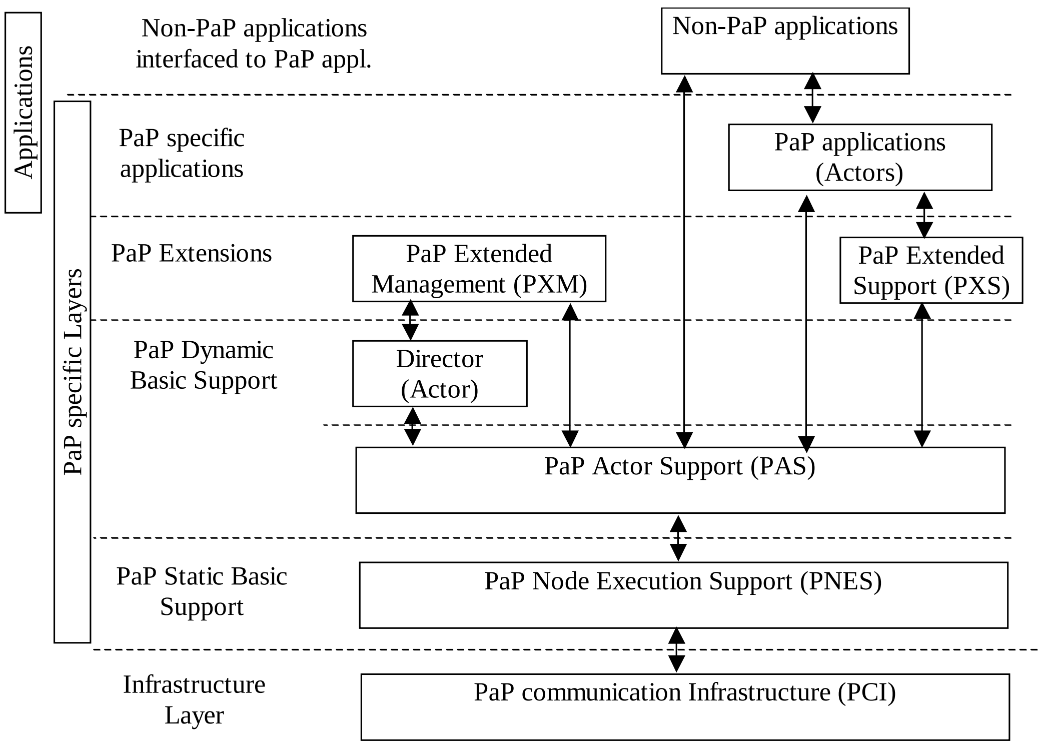


:*The Director constitutes a Director role-figure, which Role is defined by a Manuscript executed by an Actor*

Figure 1. PaP concepts

PaP support functionality: The following functions are needed: PlayPlugIn, PlayChangesPlugIn, PlayPlugOut, ActorPlugIn, ActorPlugOut, ActorBehaviourPlugIn, ActorPlay, RoleSessionAction, Subscribe, ActorChangeBehaviour, ActorBehaviourPlugOut and ChangeActorCapabilities. For details see ([Aage99], [Joha99]). The functions: ActorBehaviourPlugIn, ActorPlay and ActorBehaviourPlugOut comprise the initialisation of a generic actor pending for a manuscript, performing the manuscript, and finally making the actor pending for a new manuscript. This functionality with the addition of ActorChangeBehaviour is denoted as the basic PaP functionality. The actor is initialised by first activating its director. An actor negotiates with a director role-figure in order to obtain its behaviour. The director role-figure will create an instance of a manuscript with all necessary parameters bound particularly for the actor. The director role-figure also acts as a binding object, which helps to establish communication or interactions among actors. After receiving a manuscript from the director role-figure, an actor will start acting according to the specification described in the manuscript. From this point on in time, the actor becomes autonomous and independent of the director role-figure and constitutes an application role-figure until it terminates or wants to change its behaviour.

PaP System - Implementation Design: As illustrated in figure 1, there are two types of role-figures, the *application role-figure* and the *director role-figure*. A relation between an application role-figure and a director role-figure must always exist. Relations between director role-figures will give a possibility to obtain a distributed solution for the director role. A PaP system with more than one director needs *administrative domains* to manage the federation of responsibility between director role-figures. Figure 2 shows the structuring of the needed functionality into five layers.



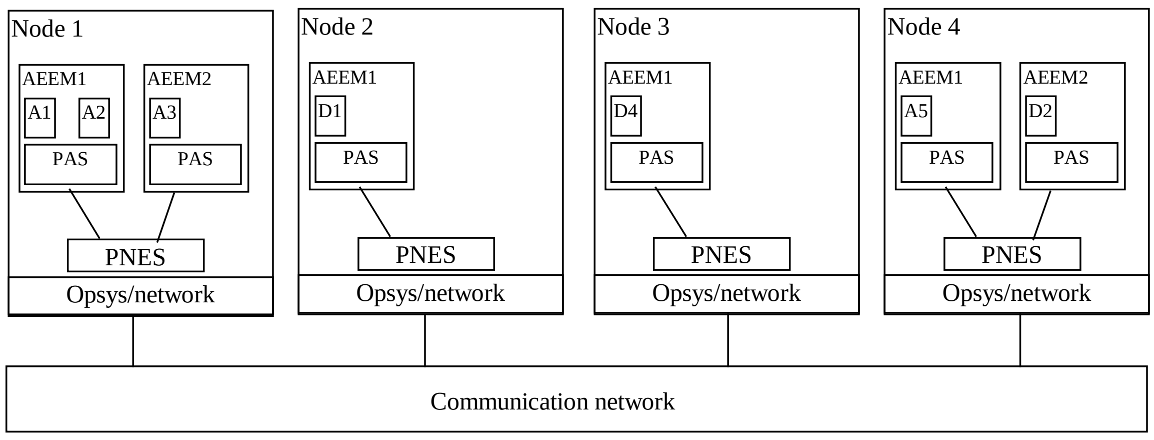
Applications
Non-PaP applications interfaced to PaP appl.
Non-PaP applications
PaP specific Layers
PaP specific applications
PaP applications (Actors)
PaP Extensions
PaP Extended Management (PXM)
PaP Extended Support (PXS)
PaP Dynamic Basic Support
Director (Actor)
PaP Actor Support (PAS)
PaP Static Basic Support
PaP Node Execution Support (PNES)
Infrastructure Layer
PaP communication Infrastructure (PCI)

Figure 2. PaP layered model

To describe the software architecture, some implementation-related concepts are needed. The most obvious hardware and software specific concepts involved are *node* and *process/thread*. A node maps directly to a computer *and a* process/thread will map one-to-one to an operating system process or thread. Figure 3 illustrates the software execution architecture or the engineering model. The Actor-environment-execution-module (AEEM) is a process/thread that executes a collection of actors with associated PaP Actor Support (PAS).

All layers in Figure 2, except for *PaP specific applications* and *non-PaP applications* are completely independent of the applications themselves. The PaP functionality will have to interface to some infrastructure technology at the bottom layer, and may interface with any type of non PaP application through the top layer.

A *PaP communication infrastructure (PCI*) architecture based on standard solutions, will usually consist of three layers with the operating system functionality (e.g. Unix or Windows) at the bottom, the network communication functionality (e.g. TCP/IP) in the middle, and some distributed system solution (e.g. CORBA ORB or Java RMI) at the top. The PCI top layer may be omitted, but that will require a more complex implementation of the interfacing module PNES if the PaP functionality require a distributed system solution.



Legend: A1 - A5: Actor1 - Actor5; A1, D2: Director1, Director2
AEEM1 – AEEM2: Actor-environement-execution-module1, Actor-environment-execution-module2

Figure 3. Example view of PaP software execution architecture

PaP Node Execution Support (PNES) makes it possible to run PaP software on a node, and for PaP functionality (i.e. executed by actors) on different nodes to interact with each other. PNES is able to receive requests from other PNESes, interpret these requests and take proper actions. PNES will also do start-up and initialisation of PASes or PCIs if that is required. PNES implements the PaP functionality that is termed the *PaP Static Basic Support* in the model. Static in this sense means that changes/extensions of the PNES functionality must be backward compatible with earlier versions because this functionality represents the "bootstrap" that is necessary to be able to run PaP applications. Only this functionality must be manually installed at a node before PaP applications can be installed and activated.

PaP Actor Support (PAS) makes it possible to create actors within the context of an operating system process/thread, to give these actors behaviour, and to communicate information between these actors and their environments. There will be one PAS instance within each Actor-environment-execution-module (AEEM) as defined above.

Director is both responsible for the management of the PaP application definitions, i.e. its part of the repertoire- and manuscript-bases, and for the management of information concerning actors, i.e. its playing-base. A director is involved in many of the functions related to the services provided by PAS.

PaP Extended Management (PXM) is additional PaP services not required for the PaP support functionality, but rather PaP extensions related to PaP operational quality. These services include functionality related to a *robust and survivable* PaP system, and a PaP system to be *QoS aware and to provide resource control*. *PaP Extended Support (PXS)* is required for the utilisation of PaP Extended Management (PXM) from actors.

PaP applications is the collection of actors implementing application role-figure. Actor instances are created using the *ActorPlugIn* function, they get their behaviour using *ActorBehaviourPlugIn* and *ActorChangeBehaviour*, they start execution using *ActorPlay,* and they terminates when using *ActorPlugOut*.

Non-PaP applications are allowed to interact with actors directly without going via the control of PAS. Such interactions can be done without the intervention of any parts of the PaP system. However, such interactions must not result into control actions that are in conflict with the responsibility of the PaP System. Non-PaP software is also allowed to use the PaP functionality supported by PAS. This possibility is actually necessary to be able to install and start the first operational PaP system. In this case the non-PaP application may interface to the same interface as used by the PaP specific applications. The non-PaP application, however, will and must perform within a separate process/thread and must be considered as one specific actor as seen from the PAS system point-of-view.

3. ACTOR MOBILITY

This type of mobility could be achieved by extending actors capabilities with movement ability, and by keeping track of this movement. In other words, the actor location specific information should be updated whenever an actor moves. In the PaP terminology we use Global Actor Identifier (GAI) as an object that maintains actor specific information regarding which PNES and PAS instances the actor executes in. Definitely, the association between actors, directors, actor-actor relations and capabilities should be dealt with carefully.

Actors move due to requirements on resource limitation, configuration change, functionality change, etc. In order to support actor mobility, the system should be able to deactivate and reactivate moving actors transparently. Therefore moved actors should be able to continue their behaviour from the point when they started to move. For this we need to define the concept of *actor state* and introduce it into plays. This aims at defining different phases of actor's behaviour and consequently the movement process will be reduced into an extended request of ActorPlugIn. As far as the implementation design is concerned, actor manuscripts should define actor's behaviour and interactions with other actors using state objects that transit from state to another. On the other hand, the movement (transition) of actors should be handled using a proper method. In the following we will introduce three method case studies that are based on replicating moved actors, using proxy to forward requests and applying a centralised agent. But before that we need to figure out what are the possibilities of actor mobility.

When actors need to move from PAS to PAS or from node to node (a node is equivalent to a PNES instance in the PaP terminology), full update of actor specific information should be applied at all instances that are related to this actor. Whenever an actor is activated it is associated to a PNES, PAS and a home director instances. PNES holds the GAI that is needed to route requests coming to this actor. PAS is the application (process or thread) in which actor objects are executing. Directors are related to actors through the definition of actors "home interface". This definition is crucial for directors to keep track of actors' location. Actor movement could be classified into three different scenarios:

1) Actor movement from PAS to PAS - in the same PNES instance
2) Actor movement from PNES to PNES – having the same director
3) Actor movement from PNES to PNES – having different directors

These scenarios are illustrated in figures 4, 5 and 6 respectively.

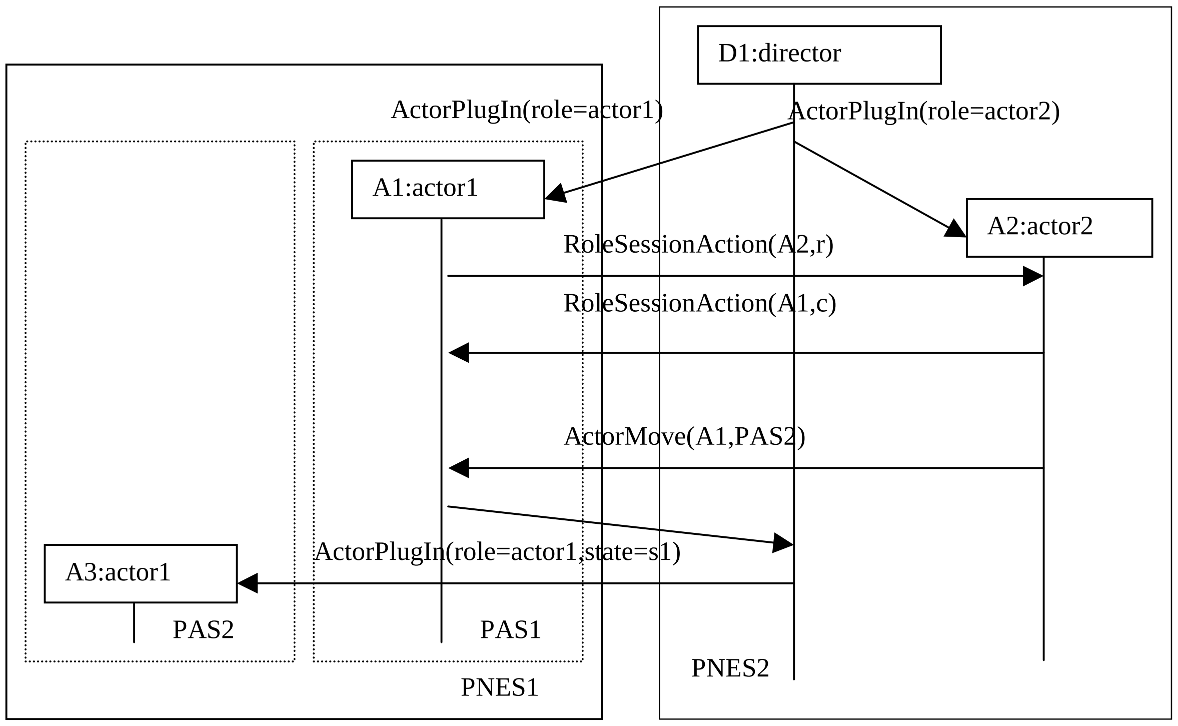


Figure 4. Actor movement of type 1.

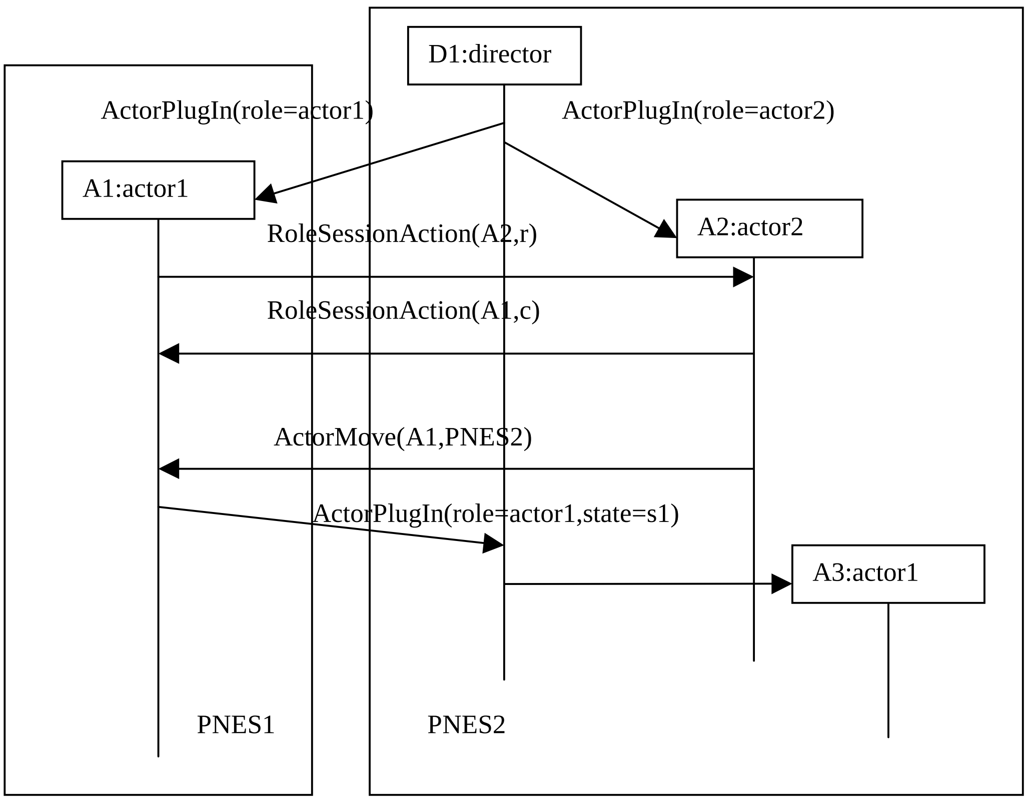


Figure 5. Actor movement of type 2.

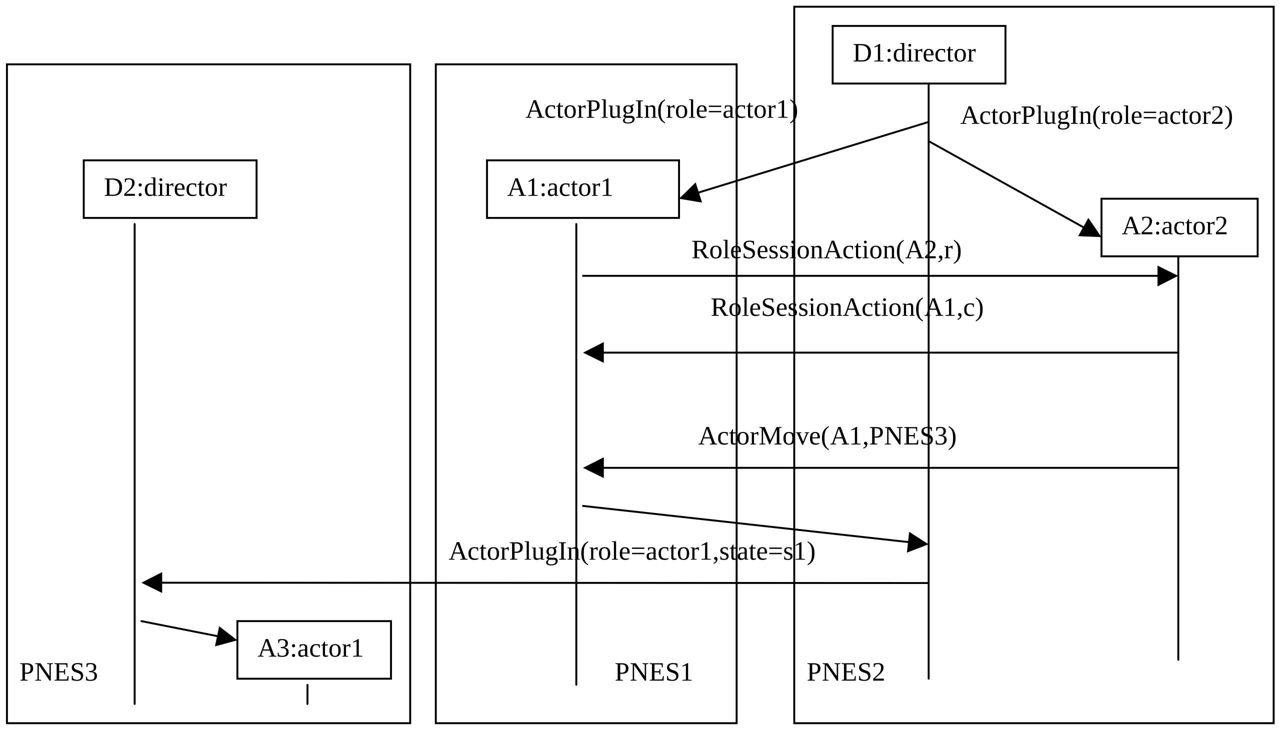


Figure 6. Actor movement of type 3.

3.1. The movement procedure

Figures 4, 5 and 6 demonstrated how actors could move in a PaP system. In all three scenarios there is an assumption that an actor is being requested to move by another actor. This is initiated by an *ActorMove* request. As we have mentioned earlier that an actor is moved by recreating it at another node or re-plugging it in, which is the case for A3 that is a new role of actor1 at different location. Note that the *ActorPlugIn* request was extended to include a parameter for *Actor State*. To carry out any of these scenarios we need to define a method to handle the various tasks associated with the management of affected PaP components. Following is a detailed description of the movement procedure.

- *The movement request*: this could be requested from within the PAS instance where the actor resides, by the director of the play or by non-PaP applications.
- *Check for possibility and applicability*: the active entity that receives the move request will check whether such an action is possible or not.
- *The move*: after receiving an acknowledgement the PAS of the actor will perform the movement method.
- *The update*: which is the last phase of this process, to update the system including directors, PNES instances, other actors, etc.

The movement method itself will be discussed in the next section. Before proceeding we need to clarify some definitions as a conclusion of the efforts done so far. The *Actor State, Active Actor* and *Actor Mobility Management* are conceptually defined to be used in any management proposal.

Actor State: Actor State is a meta-description of actor's instantaneous behaviour within a manuscript according to its role in a play. All subsequent information that an actor uses and all its interactions with other actors describe how actor's behaviour may change.

Active Actor: This is basically a notion to widen the current actor concept. An active actor is a plugged in actor that could be moving. If an actor has been replicated during the movement procedure then all of its replica should be considered as a single active actor.

Actor Mobility Management (AMM): This is meant to be the method according to which actors be moved. It is a task to be handled by the PaP support system as a whole in a decentralized or centralized fashion. All involved instances, PASes and PNESes, should update their information bases when there is an actor movement.

3.2. Actor Mobility Management methods

AMM method 1: The first method to be utilized is based on employing a single copy of a moved actor at the new location and keeping all information bases as they are. All active role sessions of that actor should be moved to the new location. Routing of requests will be done as if there was no movement at all, except for that the old actor will forward all arriving requests to the new location. This method is restricted for certain types of movement scenarios, namely 1 and 2 as described in the previous section. The transition period seems to be short and simple, however further movements of the same actor should be carried out with care. Thus instead of forwarding move requests to the new location, an *ActorPlugOut* request and a new *ActorPlugIn* request should be initiated. Figure 7 illustrate a case where an actor moves to a new location, and then moves to another location. In this figure we apply a general type of request to the actors and denote them by request1 and request2 for simplicity. Also for the other requests, e.g. *ActorPlugIn* and *ActorMove,* we don't use parameters, as well as we avoid detailed interactions with directors.

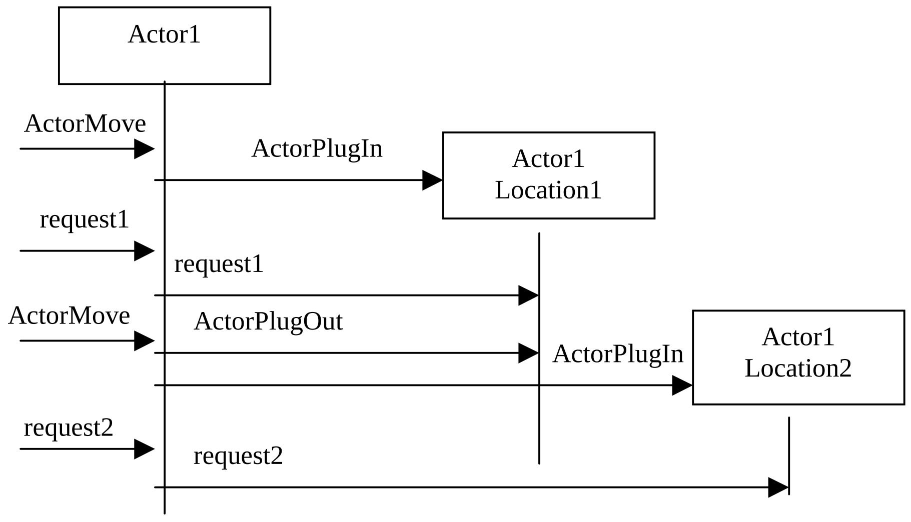


Figure 7. AMM method 1.

AMM Method 2: This is a more general method that allows actors to move freely and change "*home interface*" or associated director. Actors as objects will be created at different locations and will not be kept at their original locations. A full update of the system is required here, in which the actor identity should remain the same except for its location that need to be changed. This method seems to be more flexible from the performance point of view, but more complex in the practice. Since PaP uses a programming platform to accomplish trading and addressing such as RMI technology, relevant entity registries should be updated to cope with new locations of actors. Figure 8 demonstrates how this method works. Note that request2 is directed to the new location by the request sender.

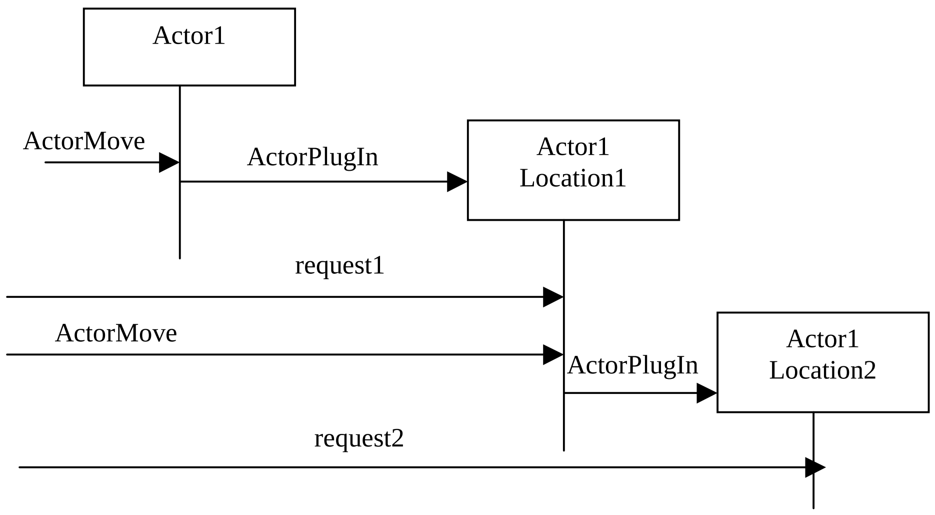


Figure 8. AMM method 2.

AMM Method 3: This method is based on centralised agent that acts as a mobility manager within the context of PaP architecture, as in figure 9. The whole PaP functionality should be modified in a way that it could include a mobility service that actors would broadcast to or consult before they move or send requests respectively. **"***LocationUpdate***"** procedure should be used prior to movements and **"***ActorDiscovery***"** procedure should be used prior to request routing. This method would be the most flexible and robust to comply with movements of actors.

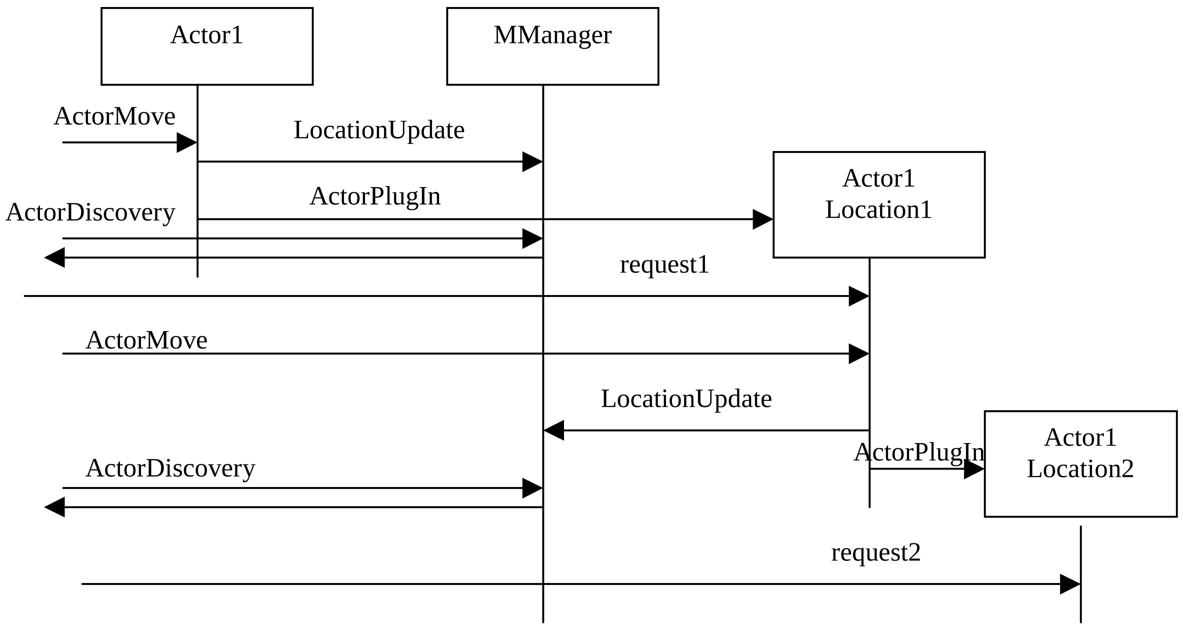


Figure 9. AMM method 3.

4. TERMINAL MOBILITY

In the PaP architectural concept there is no special definition for terminals or nodes, instead PaP components are defined to be the real world active modules that take part in the PaP functionality. By other words, PaP components in the overall PaP architecture act as the realization means of the whole concept in reality. A practical definition of terminal could be a network node that realizes the interface towards the end user. This performs the user agent, as we will discover later, and could possibly move with the user.

The mobility of a terminal or a node is defined by the change of its physical address. This is clearly seen in the engineering view of the PaP system, where at each node there is a PNES instance running at a distinct network address. To achieve mobility management for these nodes we need to keep track of their movements. So a central agent should be responsible for updating the locations of all nodes that participate in a possible PaP application. The knowledge of this central agent will be imbedded in the PNES objects, and whenever this agent moves all PNES objects should be updated. This is better demonstrated by an example. Assume first a URL address as the initial access point where the PaP support software and application are available. Assume also a terminal, possibly a laptop, which would like to access this application. The first step is to download the PaP bootstrap and modify some configuration file stating the webserver from which the application will be downloaded and a default director interface, which used to be the localhost. Once this initial phase is ready the PNES could be started from which a play will be loaded and some actors could be plugged in to perform some role figures and interact with each other. Assume now the terminal will change location, e.g. connect to the network at different place using DHCP. Note that the terminal will not be switched off. The terminal will try to resume communicating according to the behaviour of its actors. On the other hand, the rest of the communicating nodes or terminals should be able to discover this move before, while and after it is being performed.

The proposed solution is based on central agent, could be called mobility manager or MManager, which runs at a known place to all other nodes. MManager will run when the first PNES be started, and its network location should be part of the configuration file. Secondly, at each communicating node there should be a special agent performing location update and location query procedures. This agent could be called MAgent that stands for mobility agent, as in figure 10. MAgents are responsible for notifying MManager if there is a change in the node’s address. On the other hand, they should send location query on every communication event with other nodes. The MManager responds to location updates by updating the node’s location, and replies with the current location of the requested node upon receiving a location query. All in all, terminal mobility management is a kind of play that runs in the background of any PaP system that supports terminal mobility.

To obtain a clearer view of how nodes interact figure 11 shows a possible node-to-node interaction and their corresponding actors. This figure corresponds to moving node 1 in figure 10.

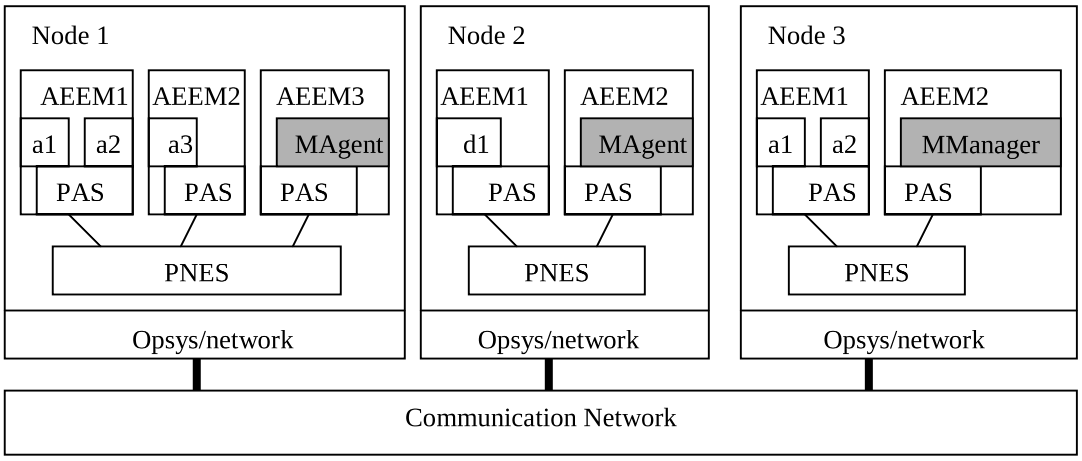


Figure 10. Engineering view of three PaP nodes with terminal mobility management.

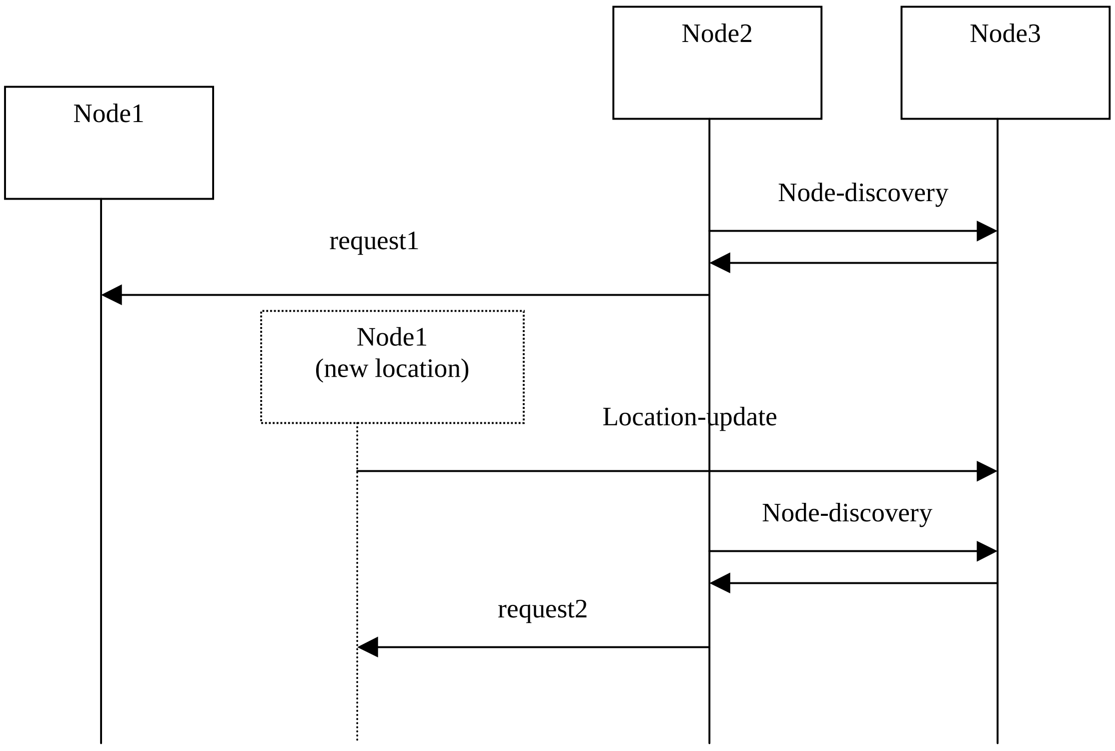


Figure 11. Node interaction before and after node moves

5. USER MOBILITY

This is also known as personal mobility, which is the utilization of services that are personalized with end users' preferences and identities independently of physical location and specific equipment. In this context we need to define the user and the services in a candidate PaP system. Let's consider for seek of generality certain service types are provided to certain users. Services are carried out by service components, which are actors. In order to achieve user mobility we need to dedicate a special actor to behave as a provider agent [Axel98]. In our proposal we suggest to assign an agent to each user, which is an actor or a set of actors carrying out system interface towards the user. This will be called User Agent (UAgent)

When we discuss user mobility we are actually talking about users and their existence in a certain domain. So services that are utilizable for a user at certain domain are not necessarily available at another domain. If a user could access all services provided to him by his home domain through another domain then these domains support user mobility. In the following we suggest a solution to achieve this.

When a user moves from his home domain into a visitor domain he will first access the PaP service provided by the visitor domain's director, or director 2 in figure 12 and 13. This service should be able to grant some users visitor status (a guest type access to basic functionality). After some initial interaction with this domain, this user will be offered to access his home domain. Meanwhile, the user is a user of the visitor domain, so he requires a UAgent. However, as this user chooses to access the services provided by his home domain then a Visitor Agent (VAgent) should be assigned to him that is controlled by the home director, director 1. VAgent has similar definition to UAgent.

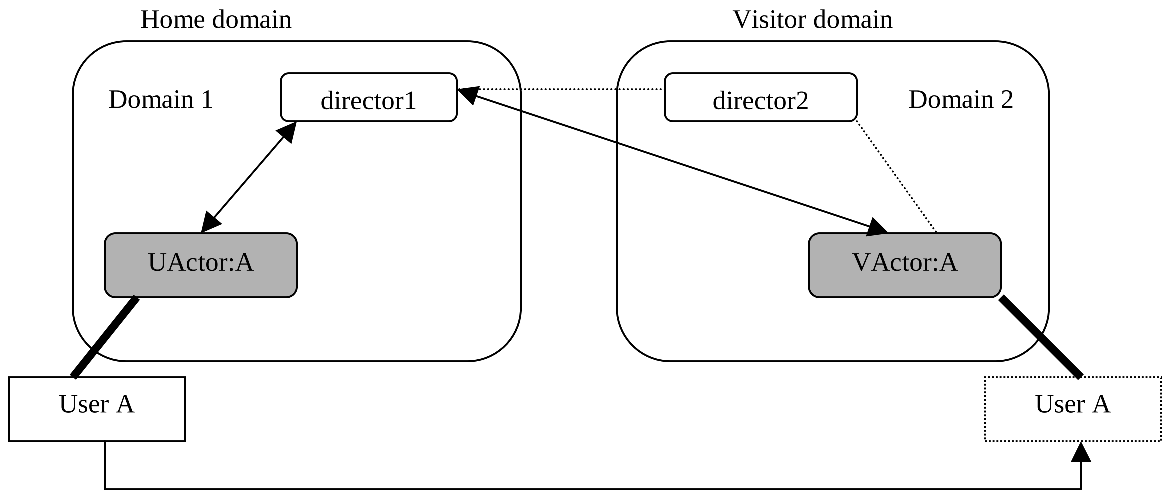


Figure 12. A proposal to handle user mobility.

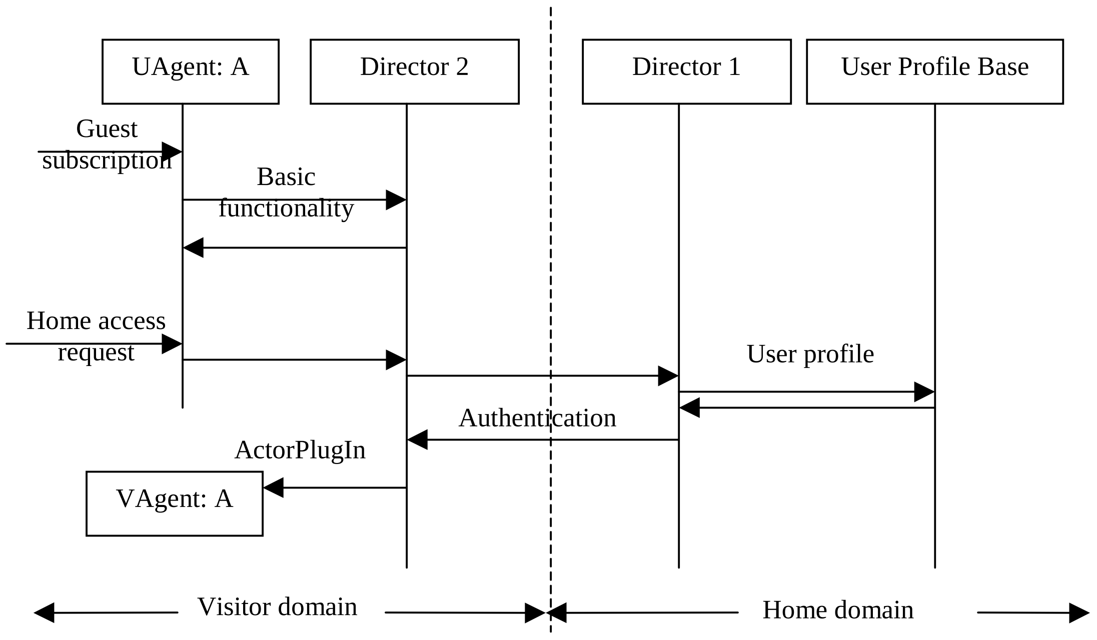


Figure 13. User mobility support: subscription, authentication and configuration.

To realize personal mobility service, personalized environment is constructed according to the user profile of the end user. According to the service architecture of TINE-C this is defined as an aggregation of four informational objects: usage context, service profile, session description and user registration. The user profile object should be moved to the visited domain in order to construct the user's environment. In our architecture this issue is less complex as we associate the VAgent to the home domain, or home director. By this way, any needed objects, e.g. the user's profile, could be accessed and downloaded from within the VAgent without any interaction with the visited domain. The main issue here is, however, at the beginning of the registration process. We should note that these domains are not business administration domains, they are instances of executed or performed plays controlled by directors.

6. SESSION MOBILITY

Session is an interaction collection who's goal is to satisfy the goal of a service by performing activities during a specific period of time. Session is also associated with the allocation of resources that are necessary to execute the aimed service. There are two types of sessions: User/Access and Service/Communication sessions.

Here we should recall the UAgent from the previous section and give it a broader definition: a computational object that manages user's preferences and performs all the operations and tasks required by the user. We should recall also that PNES object carries enough information about the node where the user is seeking access.

For the first type of session the user actor concept seems to be capable of providing this mobility. A user is granted an access to the system by assigning a UAgent. When it demands the same service from different domain a VAgent is assigned. To support the second type of session mobility we need to introduce a new role to the PaP architecture, this is the *session manager*.

We shall consider two requests to serve as a basis for session mobility: *suspend* and *resume*. A session is always initiated whenever a user is subscribed to the system and demands one of its services. The session manager is invoked and asked for a session ID. Now on, this session ID will be addressed at any future operation for session mobility. Since there is always a special actor associated with users behaviour, UAgent or VAgent, and since these actors behave according to some state transition scheme, then resuming of any suspended session is achieved by recapturing the actor's behaviour at the state where the session was stopped.

7. SUMMARY AND CONCLUSIONS

In this paper a mobility management platform for PaP network architecture was introduced. Support for Actor, Terminal, User and Session mobility is the main part of this platform. Some parts of the proposed platform are being integrated to the available demonstrator application, the tele-school application, which is based on Java RMI technology. The Actor mobility Management methods are the first parts to be integrated, as they don't require a major shift to the basic functionality. The AMM1 method in particular provides basic and yet powerful actor mobility to the PaP architecture. The idea is basically an actor proxy scheme for moving actors.

The paper dealt also with terminal mobility that is an introduction of mobile nodes. Some implementation specific details were elaborated, and various aspects of the modified architecture were mentioned. New requests and procedures will be needed to cope with this added capability. However, this addition is totally inline with the overall terminology, as we outline it as a PaP play (or service) running in the background of a PaP application. At last, we introduced a support for user and session mobility. This is mainly based on TINA definitions and its concept of a user and session. We believe our proposal for such mobility types benefits from the simplicity of mapping actors to directors in our architecture.

While in this article we laid out a platform for the integrated provision of different mobility types, there are several issues that need further investigation. On the one hand, we need to break up the overall PaP architecture into subsystems to provide selected types of mobility, which for some will be in the support and for others in the application layers. On the other hand, we need to find efficient ways to map our methods and procedures into proper applications and communicating platforms. We expect the approaches presented here to play an important role in bringing the plug and play concept to telecommunications in a more practical manner.

REFERENCES

[Aage99] Finn Arve Aagesen, Bjarne E. Helvik, Vilas Wuwongse, Hein Meling, Rolv Braek and Ulrik Johansen, Towards A Plug and Play Architecture for Telecommunications, Proceedings of IFIP SMARTNET'99, Bangkok, November 1999.

[Bies97] Andrzej Bieszczad and Bernard Pagurek, Towards Plug- and Play Networks with Mobile Code, Proceedings of ICCC'97, November 1997.

[Bies98] Andrzej Bieszczad and Bernard Pagurek and Tony White, Mobile Agents for Network Management, IEEE Communications Surveys, volume 1 number 1, 1998.

[Duts96] Joubine Dutszadeh and Elie Najm, Formal Support for ODP and Teleservices, Proceedings of the IFIP/ICCC conference on Information Network and Data Communication, June 1996.

[ITU92] ITU-T, Principles of intelligent network architecture, October 1992.

[Joha99] Ulrik Johansen, Finn Arve Aagesen, Bjarne E. Helvik and Hein Meling, Design Specification of the PaP Support Functionality, Plug-and-Play Technical Report, Department of Telematics, NTNU, 1999-12-10, ISSN 1500-3868

[Raza99] S. K. Raza and Andrzej Bieszczad, Network Configuration with Plug and Play Components, The Sixth IFIP/IEEE International Symposium on Integrated Network Management

[Tenn97] David L. Tennenhouse, Jonathan M. Smith, David Sincoskie, David J. Wetherall and Gary J. Minden, A Survey of Active Network Research, IEEE Communications Magazine, Volume 35 no 1, 1997, pages 80-86.

[TINA95] TINA Consortium, TINA-C Deliverable: Overall Concepts and Principles of TINA V1.0, February 1995.

[Axel98] Axel Kupper, User Agent – An approach for Service and User Management in 3rd Generation Mobile Networks, International Conference on Telecommunications 1998.