PLUG AND PLAY FOR TELECOMMUNICATION FUNCTIONALITY - ARCHITECTURE AND DEMONSTRATION ISSUES

Finn Arve Aagesen, Bjarne E. Helvik,
Ulrik Johansen and Hein Meling
Department of Telematics
Norwegian University of Science and Technology
N-7491 Trondheim, Norway

ABSTRACT

A plug-and-play (PaP) architecture to be applied for specification and execution of telecommunication systems functionality is presented. The architecture is based on a theatre metaphor. 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 usability of the architecture is validated through specification, implementation and testing of a PaP support system and a tele-school application demonstrator. A PaP support system that meets the flexibility and adaptability requirements and parts of the tele-school application has been implemented and validated. The implementation is based on Java RMI.

INTRODUCTION

Grade of network intelligence is here 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, 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. 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.

With dynamic PaP, the definition of individual components, and possibly the overall structure of components, can be changed on-line. One aspect of dynamic PaP is to change the services that a component provides. Another aspect is to propagate the ability to use the service to all potential service users. *The focus of this paper is on dynamic PaP, and* from now on *PaP* means *dynamic* PaP.

This paper is related to a project: PaP for Network and Teleservice Components (http://www.item.ntnu.no/~plugandplay), which *vision* is a flexible concept for telecommunication services and network functionality that can be used to increase the efficiency of, and to simplify installation, deployment, operation, management, maintenance and evolution of functionality handling software.

The general project goal is to specify, develop and experiment with parts of an architecture concept for dynamic PaP. The general research hypothesis is that an architecture concept for PaP is feasible and that it is also feasible to implement and experiment with part of such an architecture based on available software technology.

A PaP system, as discussed here, shall be: *A)*: Flexible and adaptable, *B):* Robust and survivable, and *C):* QoS aware and have resource control. In [Aage99], 9 requirements to realise the Functionality Classes *A) - C)* are specified. The PaP reference model [Aage99] is the basis for realising the Functionality Class A), which further is a basis for realising the Functionality Classes B) and C). The Class A requirements as well as the PaP reference model is presented in the next section. Within the Functionality Classes B) and C) research work is going on within the following four areas 1): PaP for Teleservices [Jacq2000], 2): Dependability and Intrusion Avoidance in PaP systems, 3): PaP for Mobile Components, and 4): Capability Handling in PaP systems.

A construction model for a software system that realises Property Class A [Joha99a], has been implemented using Java RMI [Joha01], and a PaP demonstrator based on a Tele-school application has been specified and partly implemented [Joha99b]. The purpose of this demonstrator is both to demonstrate the realisation of the Property Class A, and to be the fundament of a testbed for the research on the Functionality Classes B and C. The goal is a PaP support system which implements all the Property Classes A)-C).

The objectives of this paper is to present our basic conceptual framework for handling dynamic PaP and also the demonstrator application to be used for the validation of the conceptual framework. A functional model denoted as *the PaP reference architecture* is first presented. Thereafter a *design architecture* supporting the implementation of the PaP reference architecture is presented. The *tele-school application* is then presented, before finally giving *summary and conclusions*.

A PAP REFERENCE ARCHITECTURE

PaP components

The entities in the system subject to PaP are the *PaP components*, which 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. Pure hardware modules are not feasible in the context of dynamic PaP. PaP components will coexist with components that do not have the PaP functionality. These are denoted as non-PaP components. The properties of Functionality Class A, which are related to flexibility and adaptability require:

1) a system structure and functionality that is not fixed (adding, moving, removing components and changing component functionality according to needs and capabilities),
2) that new components, their external services capabilities and needs are found automatically (awareness of new components and capabilities, propagation of needed information about changes, propagation of needed new functionality),

3) a continuous adaptation to the environment and operation strategies/policies (new component functionality, new teleservices, new service and network management functionality, new policy functionality),
4) containment and aggregation.

The functional object model to be defined in the following subsection aims to be a basis for satisfying these requirements.

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 is the *computational* and the *engineering* viewpoints. The PaP components are basically engineering viewpoint objects. The PaP components have a computational viewpoint specification by the PaP functional objects, which are basically computational viewpoint objects. The computational model will also model the information which is subject to dynamic changes caused by the behaviour. Information models are supplementary models supporting the behaviour models.

Most object-oriented systems supports dynamic creation and removal of individual object instances. While this may be sufficient for static PaP, dynamic PaP requires in addition that:

- it is possible to change the definition of object instances and object instance structures, i.e. to change their type,
- to propagate the effect of such changes to involved object instances.

Thus, dynamic PaP requires a PaP support system with the ability to manipulate type definitions, and to dynamically change object behaviours and object structures according to the changes of the corresponding types. This situation has many similarities with the theatre, which is chosen as a model to describe the PaP functionality of the system. The relationship between important PaP concepts is shown in Figure 1, while Figure 2 illustrates the basic structure of a PaP system.

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 is defined by *roles* and the role is formalised my a *manuscript*. These concepts have meaning similar to as these concept 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. An actor is able to play various *roles*. Different from a theatre, and caused by the nature of telecommunication service providing systems, an actor can have its behaviour related to various plays at a time. However, an actor performs only one manuscript at a time. A PaP *component*, however, can handle various manuscripts by using various actors playing different manuscripts.

A manuscript defines the *entire* behaviour of an actor. A *role-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). The entire role can be composed by logically joining the role-sessions. The manuscript both specifies the

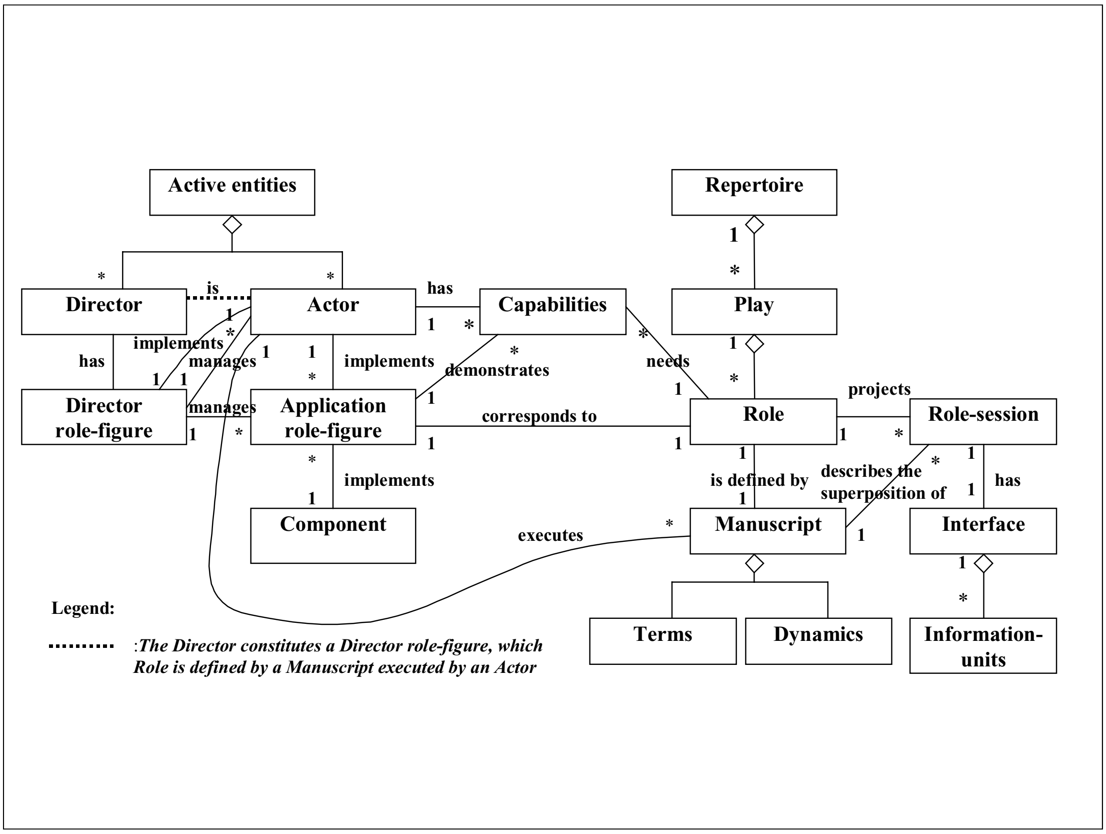


Figure 1. PaP concepts

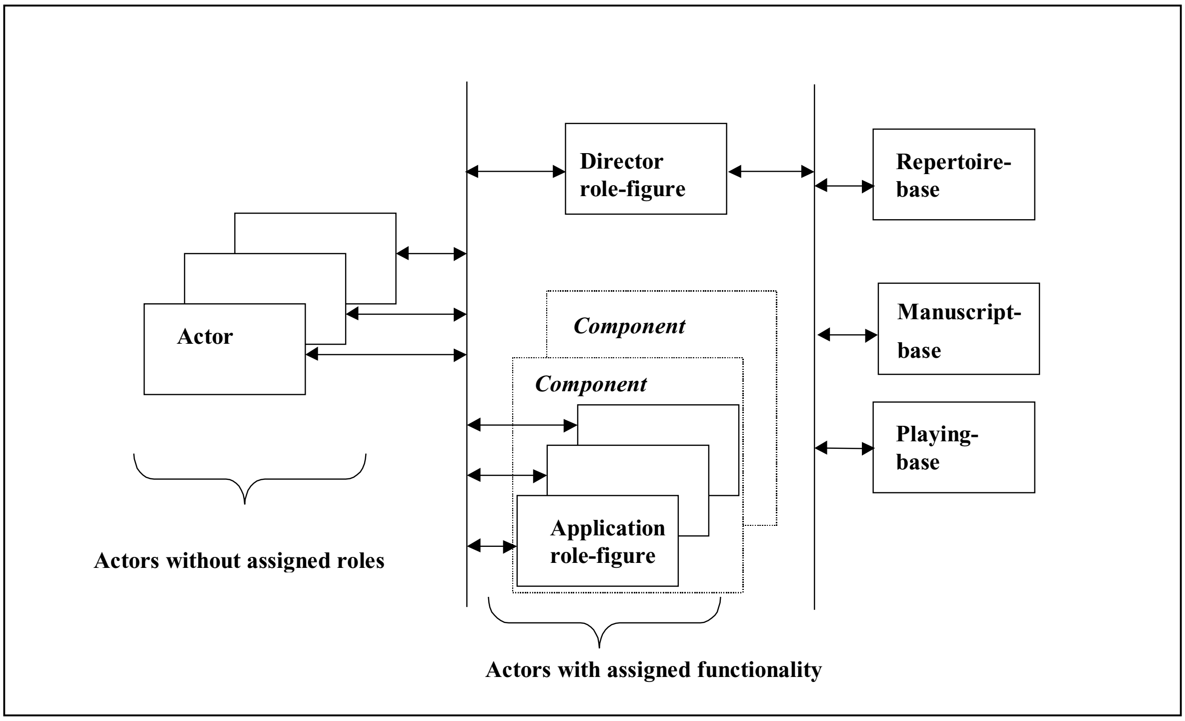


Figure 2. PaP system - Basic structure

cooperating PaP objects, the interactions with the cooperating PaP objects and internal behaviour resulting from an incoming interaction.

An actor also has a defined set of *capabilities*, which is the ability or power to do something. Capabilities are inherent properties of an actor, which can not be removed, replaced or copied without removing, replacing or copying the entire actor. 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.

Capabilities also encompass proprietary information and authorization as found in agents. In other words, an actor is a generic abstraction of the whole or part of the functionality of a real-world PaP component as defined above. The actors capability will define which real-world PaP component functionality it is able to act on behalf of. Capabilities are further discussed in [Aage99].

A *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*. A 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 initiative to the plug-in of an actor may come from a role-figure

with another role than the role that is going to be played by the actor that is to be plugged in, i.e. the initiative can come from an actor playing the same or another role. The initiative to plug-out can come from the same role-figure as the plug-out role-figure. The director role-figure guides actors in the plug-in phase as well as in the plug-out phase. Important functionality related to the plug-in-phase is *actor identification, actor access control, actor capability control* and *actor resource and QoS negotiation and allocation*.

PaP components are realised by actors, and actors are the entities constituting instances (application role-figures) of the PaP functionality objects. Figure 2 illustrates a system with some actors without assigned roles, one director role-figure, several components realised by application role-figures, one instance of a *repertoire-base*, one instance of a *manuscript-base* and one instance of a *playing-base*. For simplicity the system illustrated is a centralised system. The *manuscript-base* has the manuscripts used by the actors to play their roles. The *playing-base* keeps a structural model of the instances of PaP objects that is actually playing. The *repertoire-base* keeps an overview of the potential plays and roles. Actors get an instance of a manuscript from the manuscript-base via the director. The manuscript of the director is also a part of the manuscript-base.

PaP support functionality

The following functions are needed: *PlayPlugIn, PlayChangesPlugIn, PlayPlugOut, DynamicDetectionOfNeedsForActors/Plays/Roles, ActorPlugIn, ActorPlugOut, ActorBehaviourPlugIn, ActorChangeBehaviour, ActorBehaviourPlugOut, ActorPlay, Subscribe, RoleSessionAction* and *ChangeActorCapabilities*. For details see ([Aage99], [Joha99a]).

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 want to change its behaviour.

PAP SYSTEM - IMPLEMENTATION DESIGN

As already explained, 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 3 shows the structuring of the needed functionality into five layers.

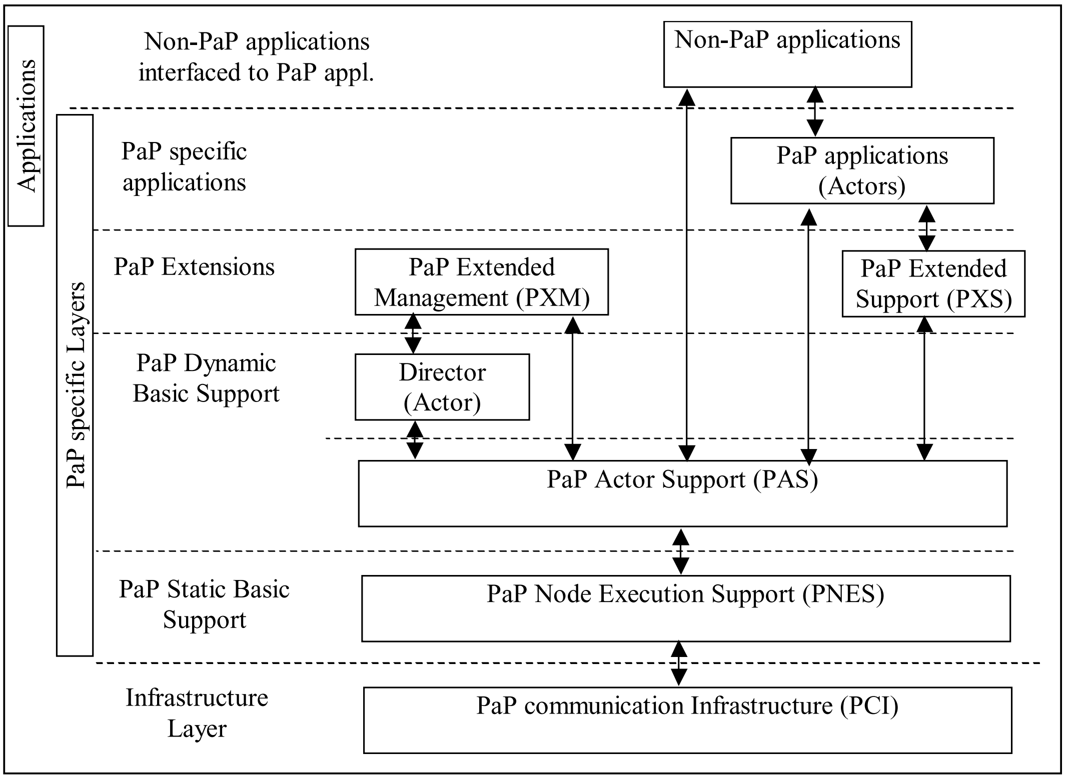

Applications
PaP specific Layers
Non-PaP applications interfaced to PaP appl.
Non-PaP applications
PaP specific applications
PaP applications (Actors)
PaP Extensions
PaP Extended Management (PXM)
PaP Extended Support (PXS)
PaP Dynamic Basic Support
Director (Actor)
PaP Actor Support (PAS)
PaP Static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

Figure 3. 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 4 illustrates the software execution architecture. The Actor-environment-execution-module (AEEM) is a process/thread that executes a collection of actors with associated PaP Actor Support (PAS). A collection of actors is here one or more actors constituting application role-figures or director role-figures.

All layers in Figure 3, 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 NT) 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.

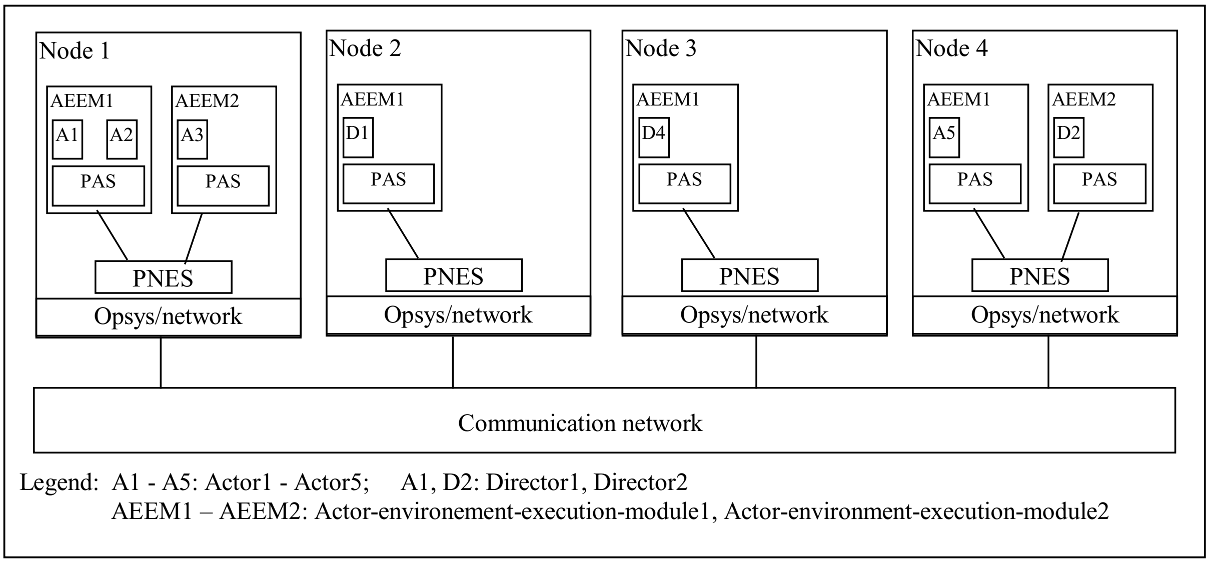


Figure 4. Example view of PaP software execution architecture

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.

THE TELE-SCHOOL APPLICATION

The specification of the demonstrator is based on the need to validate all aspects of the basic PaP functionality, some teleservice functionality for wireless applications and some functionality to satisfy some of the requirements of the Functionality Classes B and C.

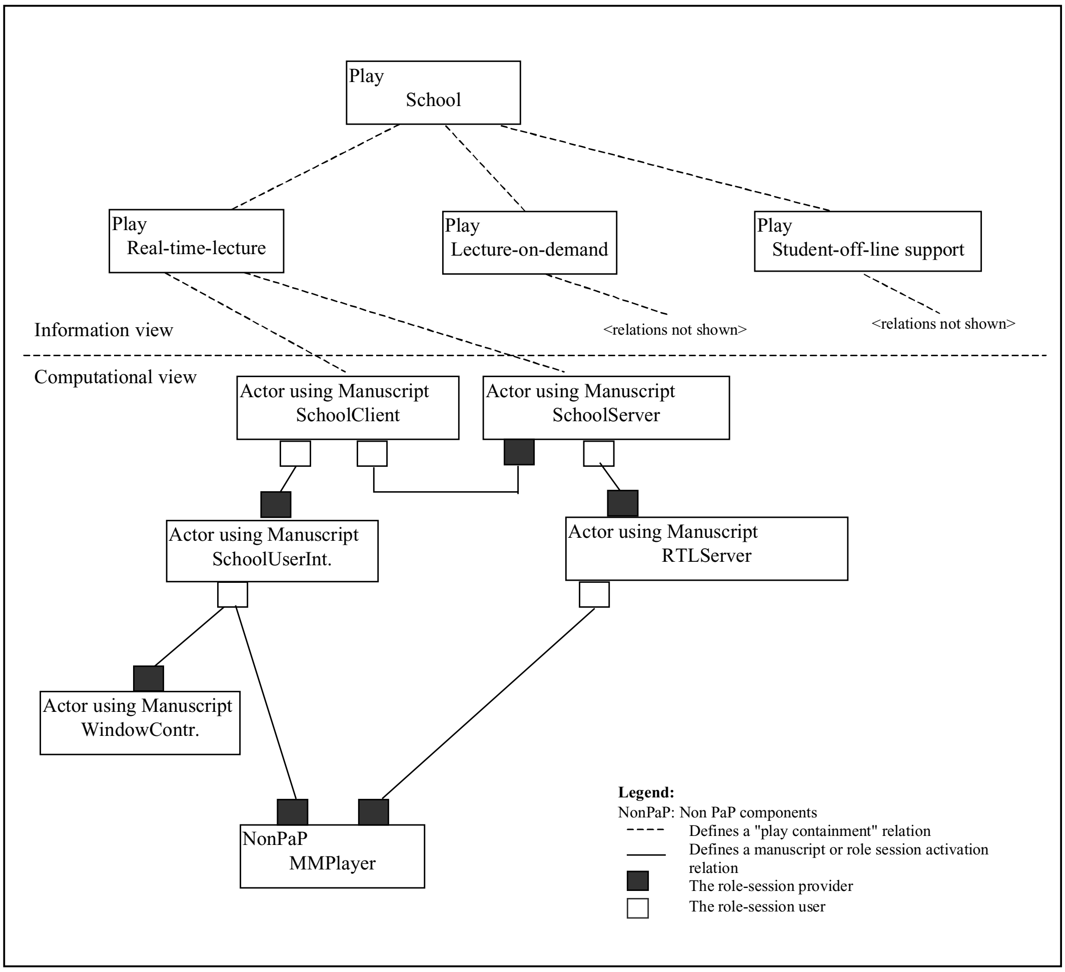

Play School
Play Real-time-lecture
Play Lecture-on-demand
Play Student-off-line support
<relations not shown>
<relations not shown>
Information view
Computational view
Actor using Manuscript SchoolClient
Actor using Manuscript SchoolServer
Actor using Manuscript SchoolUserInt.
Actor using Manuscript RTLServer
Actor using Manuscript WindowContr.
NonPaP MMPlayer
Legend:
NonPaP: Non PaP components
---- Defines a "play containment" relation
—— Defines a manuscript or role session activation relation
■ The role-session provider
□ The role-session user

Figure 5. Example real-time lecture architecture

Tele-school is an application to be used as a demonstrator. Its main concepts are *school*, *school application manager*, *students*, *teachers*, *courses* and *lectures*. Basic *communication modus* between teachers and students, where teacher and students physically may be located in different geographical areas are: *interactive dialogue* (*chat* type service), *addressed messaging* (E-*mail* type service), and *non-addressed messaging* (*news* type service).

The multimedia components *audio*, *video* and *text* shall be used for the *content* representation. Multimedia PC type equipment shall be used for interaction between teachers/students and the application system. The tele-school services provided are *real-time lecture*, *lecture on demand* and *students off-line support*. The first is the real-time lecture performance by a teacher to attached students. Students may ask questions and get answers from the teacher. Lecture on demand offers the possibility for students to go through already performed lectures that has been "electronically recorded". Students may freely select when to go through the lecture and they also may ask questions to the teacher, which will answer question when available by using the Student off-line support functionality. Figure 5 shows role-figure and role-session structure for a real-time lecture.

Behaviour description for the functional design

First regard the example message sequence chart describing a small part of the tele-school application, shown in Figure 6.

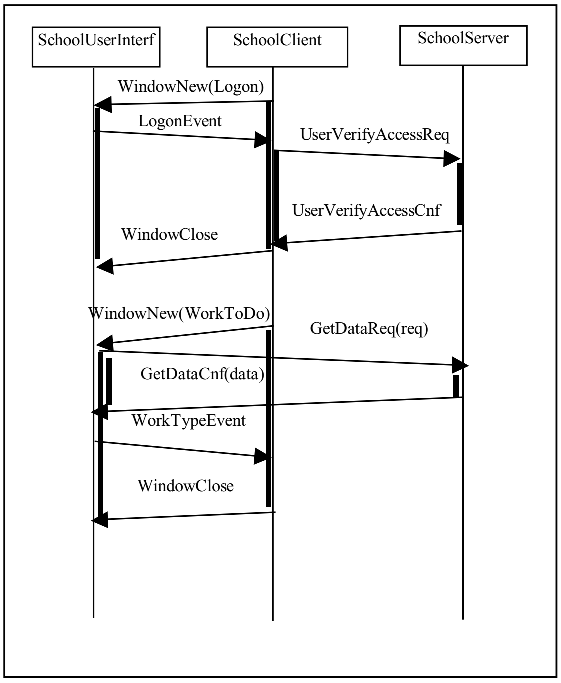


Figure 6. Example interactions between application role-figures in the tele-school

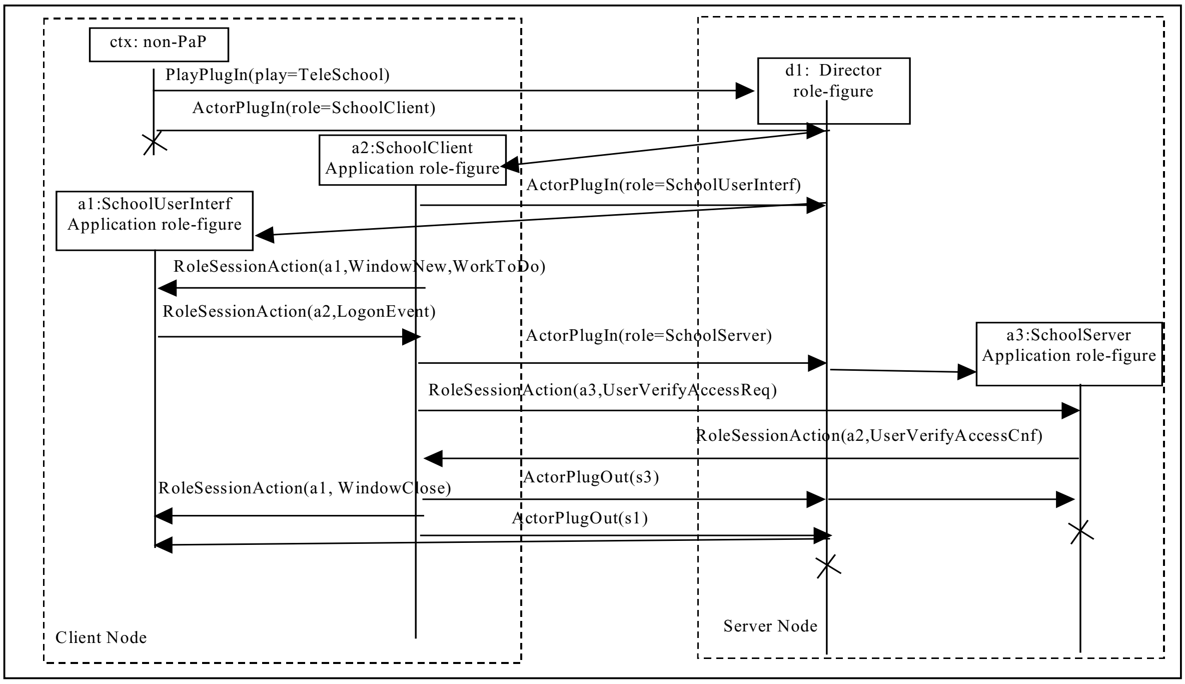
The *SchoolUserInterf, SchoolClient* and *SchoolServer* has been identified as three roles to be played, and the functionality shown in the figure is related to user system log-on, access verification and the user selection of what type of work to do. This is a part of the real-time lecture functionality. A set of information elements (e.g. *WindowNew* and *WindowClose*) to be used for role-figure interactions has been identified. Role-sessions are specified as well defined sequences of interactions between two role-figures and are indicated by the additional vertical lines in the figure. The first identified role-session used by the roles *SchoolUserInterf* and *SchoolClient* contains the information elements *WindowNew, LogonEvent* and *WindowClose*.

The *SchoolClient* role has two role-sessions active at the same time, one with the *SchoolUserInterf* and one with the *SchoolServer*.

Integration with PaP functionality

Figure 6 shows some application specific aspects of the tele-school, in addition to the identification of roles and role session examples. So how does the PaP specific concepts and functionality apply together with the tele-school application? The PaP functionality is generic, but it is important to be aware of how it is integrated into the application, since the application designers must consider both the application and the PaP functionality.

Figure 7 extends Figure 6 by showing the mapping from roles to actors, the use of PaP functionality in application execution, and how an application become available for use. Note that an application-role does not contain the interaction with the director role-figure. This interaction is a part of the generic actor behaviour. Figure 7 shows that each of the three roles defined in Figure 6 are played by separate actors, named *a1, a2* and *a3* forming the corresponding role-figures.



ctx: non-PaP
PlayPlugIn(play=TeleSchool)
d1: Director role-figure
ActorPlugIn(role=SchoolClient)
a2:SchoolClient Application role-figure
ActorPlugIn(role=SchoolUserInterf)
a1:SchoolUserInterf Application role-figure
RoleSessionAction(a1,WindowNew,WorkToDo)
RoleSessionAction(a2,LogonEvent)
ActorPlugIn(role=SchoolServer)
a3:SchoolServer Application role-figure
RoleSessionAction(a3,UserVerifyAccessReq)
RoleSessionAction(a2,UserVerifyAccessCnf)
ActorPlugOut(s3)
RoleSessionAction(a1, WindowClose)
ActorPlugOut(s1)
Client Node
Server Node

Figure 7. Example interactions between actors in the tele-school application

In addition there exists another actor, *ctx*, which has the responsibility of tele-school application installation and start-up, and an actor *d1*, which serves as director role-figure for all actors which constitutes application role-figures. The application has been distributed on two nodes, namely a *client node*, which runs the client specific application part, and a *server node*, running the parts of the application common to multiple tele-school clients. The actor implementing the director role-figure has been selected to perform on the server node.

The MSC example shows that the tele-school application has not yet been installed and is therefore installed by the *ctx* actor using *PlayPlugIn(play=TeleSchool)* prior to starting a client part of the application by using *ActorPlugIn(role=SchoolClient)*. *PlayPlugIn* is the PaP function used to install play definitions into the manuscript-base which is managed by the director role-figure. Play definitions may be changed by *PlayChangesPlugIn*, and removed by using *PlayPlugOut*.

Role-sessions are created by *ActorPlugIn* and terminated by *ActorPlugOut*. In addition *ActorPlugIn* will imply the creation of an actor instance if no existing actor is able to satisfy the requirements to cooperating actor considering location, role and capabilities specified by the *ActorPlugIn* parameters. *ActorPlugOut* will destroy an existing role-session between two role-figures, and may imply the termination of an actor if that has been specified as a parameter to *ActorPlugOut* or if the actor's termination condition specifies so.

Figure 7 indicates that the director is involved in the execution of many of the PaP specific functions such as *PlayPlugIn, ActorPlugIn, ActorPlugOut*. This is necessary, because the director serves, not only the manuscript-base, which is as a repository for play specifications, but also serves the playing-base, which is a repository for existing actor instances and the role-figures constituted. This means that, as seen from actors, the director become a central server. The PaP specific function *RoleSessionAction*, which is used for communication on established role sessions, does not involve the director.

Two levels of plug-in/plug-out is illustrated. First, the plug-in of the specification of the PaP application functionality (i.e. component installation) done by the *PlayPlugIn*, then the plug-in/out of the applications themselves (i.e. component execution) done by *ActorPlugIn/ActorPlugOut*. Note that *ActorPlugOut* does not necessarily mean that the associated actor terminates. It is the interaction sequence, i.e. the role-session between two application role-figures that is terminated.

Not shown in the figure is the possibility to dynamically change both *component definitions* and *actors behaviour* for an operational system by using *PlayChangesPlugIn* and *ActorChangeBehaviour*, respectively. Procedures have been defined for dealing with these changes that may result from actor requests.

SUMMARY AND CONCLUSIONS

A dynamic PaP architecture concept, as well as an implementation design of this architecture concept, has been presented. The objective of this work is to simplify and speed up the tasks of deployment, installation, operation, management, maintenance and evolution of software related to telecommunication equipment and services functionality. A tele-school application to be used for demonstration and validation of the various elements of the plug-and-play architecture has also been presented.

The PaP support system that meets the flexibility and adaptability requirements and parts of the tele-school application has been implemented and validated. The implementation is based on Java RMI. We feel that our approach has lead to a powerful solution for the handling of adaptability and flexibility, and that it is a suited basis for the extension of the functionality to comprise the functionality classes B and C, which comprise robustness and survivability, and QoS awareness and resource control, respectively. These extensions are at the moment subjects for ongoing research.

The dynamic PaP architecture is supporting PaP components which are real-world concrete active hardware and software modules. The PaP component functionality is defined by a PaP functional object model, consisting of functional PaP objects. A functional PaP object is an instance of a PaP object type. Dynamic PaP requires that it is possible to change the behaviour of an object and to propagate the effect of such changes. A functionality analogous to the theatre is chosen as a basis for a reference architecture for PaP. The most central concepts are actor, director, role, role-figure, play, manuscript and capability. An actor's capabilities define the possibilities for playing various roles according to manuscripts and then to constitute various role-figures.

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.

[Jacq00] Jacqueline Floch and Rolv Bræk, Towards Dynamic Composition of Hybrid Communication Services, IFIP TC6 Sixth International Conference on Intelligence in Networks, Vienna, September 2000.

[Joha99a] 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

[Joha99b] Ulrik Johansen, Finn Arve Aagesen, Bjarne E. Helvik and Hein Meling, Demonstrator - Requirements and Functional Description, Plug-and-Play Technical Report, Department of Telematics, NTNU, 1999-12-10, ISSN 1500-3868

[Joha01] Ulrik Johansen, Plug-and-play – Software Design, Implementation and Use. Plug-and-Play Technical Report, Department of Telematics, NTNU, 2001-02-10, 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.