NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY, MATHEMATICS AND ELECTRICAL ENGINEERING
DEPARTMENT OF TELEMATICS

Mobility support for wireless devices - within the TAPAS platform

Master thesis by
Eirik Lühr

Trondheim, January 2004

Preface

This report is the result of my master thesis carried out at the Norwegian University of Science and Technology (NTNU) in Trondheim, during the fall of 2003. The master thesis was suggested by my advisor, Mazen Malek Shiaa, and is a continuation a project assignment which was carried out during the spring of 2003.

The time spent on this project has been very interesting and meaningful, and I have learnt a great deal about this particular field of study – mobility. The TAPAS architecture is a great undertaking, and the learning curve was very steep. Nevertheless, I feel that I have not only learnt the fundamental basics of the architecture but also a deeper understanding of its inner workings.

First of all I would like to thank my advisor, Mazen Malek Shiaa, for all the time he have spent explaining and outlining the various aspects of the TAPAS platform, and for helping me along whenever the need arose – thank you! I am also thankful for the care and support that my family always has given me – mom, dad and Eivind. Last, but not least, a big thankful hug to my always supportive girlfriend, Anne!

Oslo, 11 January 2004

Eirik Lühr

Abstract

Telematics Architecture for Plug-and-play Systems, or TAPAS, is a software architecture that facilitates for dynamic introduction of new distributed services, or upgrading of existing ones, in a communication network. The Department of Telematics, in cooperation with SINTEF, have implemented a prototype of this architecture using the Java programming language. A lightweight version of TAPAS has been specified and developed, and is aimed at deployment on small handheld wireless devices – personal digital assistants (PDA).This report's main area of focus lies in designing and implementation of TAPAS Extended Management (TXM) and TAPAS Extended Support (TXS) functionality for handling terminal and actor mobility for the MicroTAPAS support architecture. In addition, a test application is to be developed, and testing is to be performed.

Chapter one introduces the reader to the concepts behind TAPAS and MicroTAPAS, as well as mobility support. A short overview of relevant previous papers on TAPAS and mobility is also presented. Chapter two presents a literature review over related technologies.

In chapter three, the mobility support for wireless devices are presented, and starts with a list of definitions, before moving to the framework, and lastly presents terminal and actor movement. The implementation of the specified mobility extensions are discussed in chapter four, along with various diagrams, data files and various additions and enhancements.

The test application is presented in chapter five, and is used in testing the implemented functionality in chapter six. The three last chapters, seven, eight and nine presents the faced challenges and solutions, suggested improvements and further work, and, finally, the conclusion.

Table of contents

Table of figures and tables

1 Introduction

In this introductory chapter, four important topics will be discussed. The chapter starts with an introduction to the Telematics Architecture for Plug-and-Play Systems (TAPAS) and the main ideas and concepts behind it. Secondly, an introduction to a version of TAPAS aimed at wireless handheld devices, MicroTAPAS, and the motivation behind this support system will be given. Thirdly, the notion of mobility is introduced, along with a quick overview of what this thesis is all about. The last section summarizes some of the previous work on mobility related to TAPAS.

1.1 Introduction to TAPAS

The Telematics Architecture for Plug-and-Play Systems (TAPAS) is a research project at NTNU which aims at developing and architecture for network-based services systems with a) flexibility and adaptability, b) robustness and survivability, and c) quality of service (QoS) and resource control. The goal is to enhance the flexibility, efficiency and simplicity of system installations, deployment, operation, management and maintenance by enabling dynamic configuration of network components and network-based service functionality. Four main architectures have been developed – the basic architecture ([JOHU1], [AAGF1]), mobility handling architecture ([MALM1], [MALM2], [MALM3], [LILL]), dynamic configuration architecture ([AAGF2], [AAGF3]), and the adaptive service configuration architecture [AAGF3].

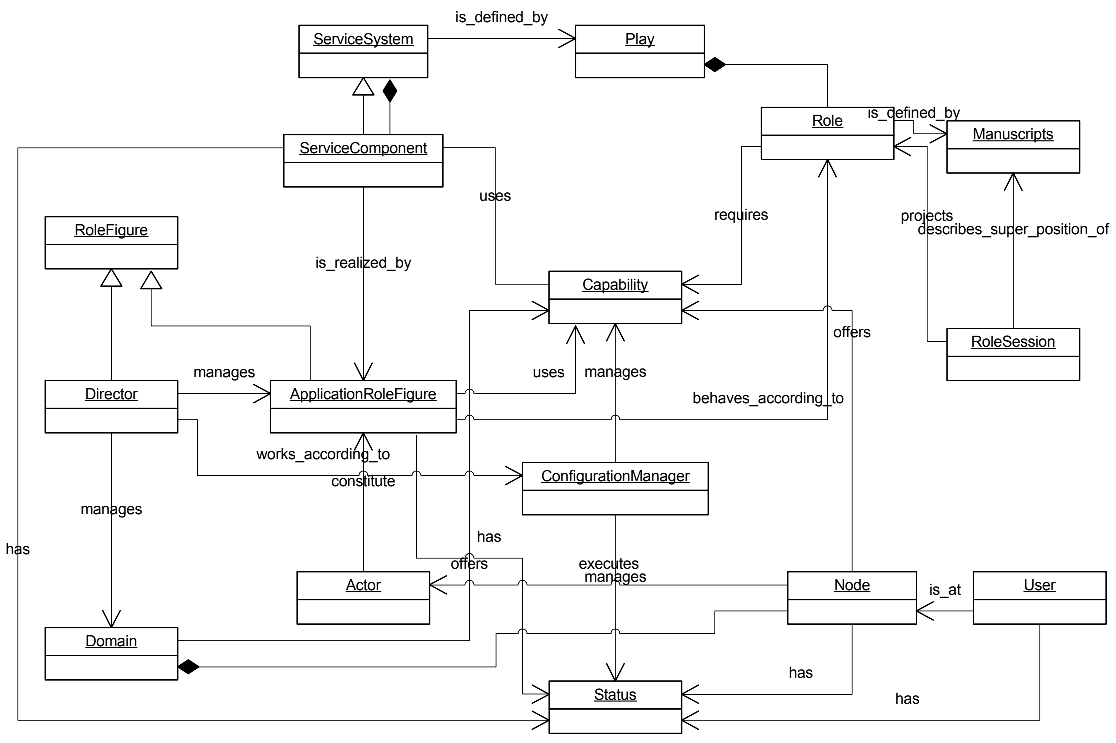


Figure 1-1: TAPAS basic architecture (Object Model)

The TAPAS basic architecture, illustrated in Figure 1-1, is based on generic *Actors* (software components) in the *Nodes* of the network that can download *Manuscripts* defining *Roles* to be played, each representing different functionality. *Nodes* may be servers, routers and switches, and user terminals, such as telephones, laptops, PCs, PDAs, etc. The model is founded on a theatre metaphor, as illustrated by Figure 1-2 [AAGF1, page 7], where *Actors* perform *Roles* according to predefined *Manuscripts*, and a *Director* manages their performance (i.e. their plug-in and plug-out phases), and also a *Director* represents a *Domain*. *ServiceSystem* consists of *ServiceComponents*, which are units related to some well-defined functionality defined by a *Play*. A *Play* consists of several *Actors* playing different *Roles*, each possibly having different requirements on *Capabilities* and *Status* of the executing system. A *RoleSession* is a projection of the behavior of an actor with respect to one of its interacting *Actors*. *Capability* is an inherent property of a *Node*. The ability of *Actors* to play *Roles* depends on the defined required *Capability* and the matching offered *Capability* in a *Node* where they intend to execute. *ConfigurationManager* is responsible for obtaining a snapshot of all system resources, and taking decisions on where and how *Capabilities* and *Actors* may be installed and executed. *Capabilities* may be resources (e.g. CPU, hard disk, transmission channels), functions (e.g. printing, encryption devices), or data (e.g. user login, access rights).

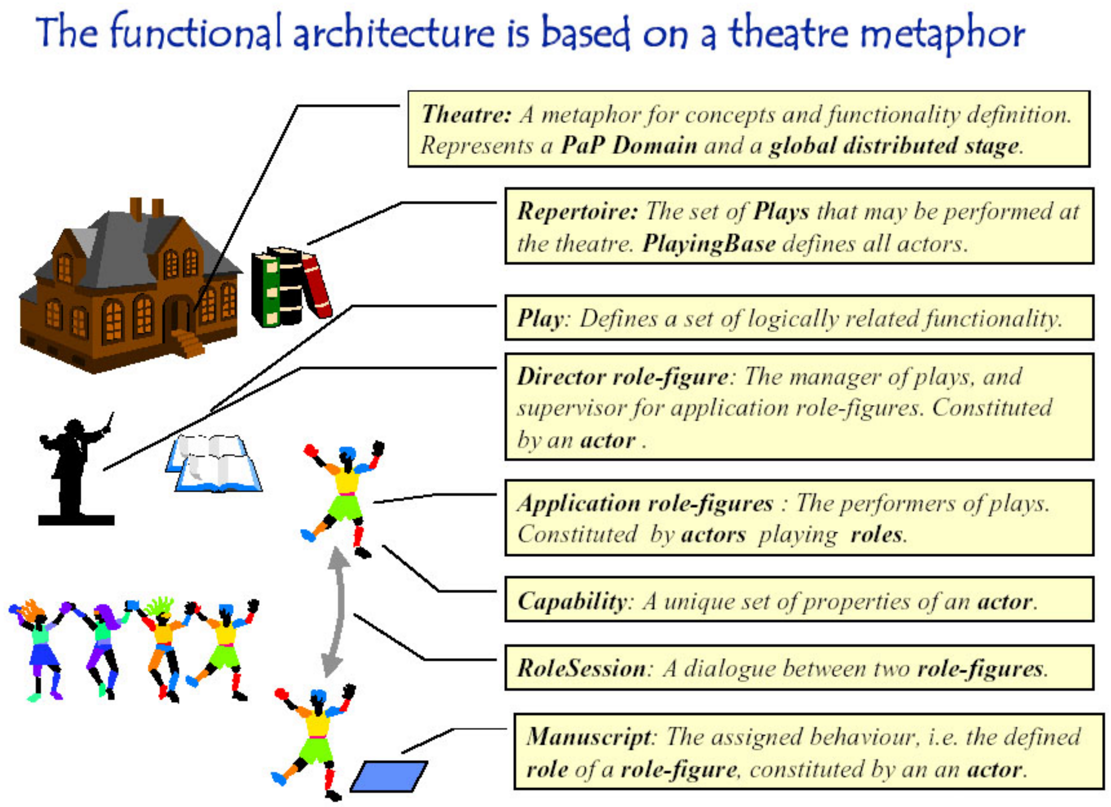

The functional architecture is based on a theatre metaphor
Theatre: A metaphor for concepts and functionality definition. Represents a PaP Domain and a global distributed stage.
Repertoire: The set of Plays that may be performed at the theatre. PlayingBase defines all actors.
Play: Defines a set of logically related functionality.
Director role-figure: The manager of plays, and supervisor for application role-figures. Constituted by an actor .
Application role-figures : The performers of plays. Constituted by actors playing roles.
Capability: A unique set of properties of an actor.
RoleSession: A dialogue between two role-figures.
Manuscript: The assigned behaviour, i.e. the defined role of a role-figure, constituted by an an actor.

Figure 1-2: TAPAS theatre model

Figure 1-3, as presented in [MELH1, page 15], gives an overview of the various layers in the TAPAS architecture. Each layer contains different support functionality and has by [MELH1, page 16-17] been summarized as follows:

- **TAPAS Communication Infrastructure (TCI)[1]:** TCI uses Java/RMI and ‘rmiregistry’ for communication between the various nodes which constitutes a TAPAS domain.

[1] Formelry known as PaP Node Communication Infrastructure (PNCI), as TAPAS was named Plug-and-Play (PaP).

- **TAPAS Node Execution Support (TNES)[2]:** This layer makes it possible to run TAPAS on a node, and facilitates routing of communication and other essential tasks.
- **TAPAS Actor Support (TAS)[3]:** Makes it possible to create and execute actors within the context of an operating system process. Additional functionality is routing between actors and TNES instances. Each TAS instance is a separate Java Virtual Machine (JVM) instances.
- **TAPAS Director:** The Director is responsible for the management of plays, manuscripts and actors for its own TAPAS domain and interacts with application actors.
- **TAPAS Extended Management (TXM):** Support of extended services not required for TAPAS support functionality, but to satisfy specified operational properties and requirements.
- **TAPAS Extended Support (TXS):** Required for the applications ability to utilize TXM functionality.
- **TAPAS Applications:** The collection of application actors. Instances created/removed by using ActorPlugIn/ActorPlugOut support functions. Interfaces TAS.
- **Non TAPAS applications:** Functionality not defined according to Application actor requirements, but is allowed to communicate with actors, and to utilize TAPAS Support functionality.

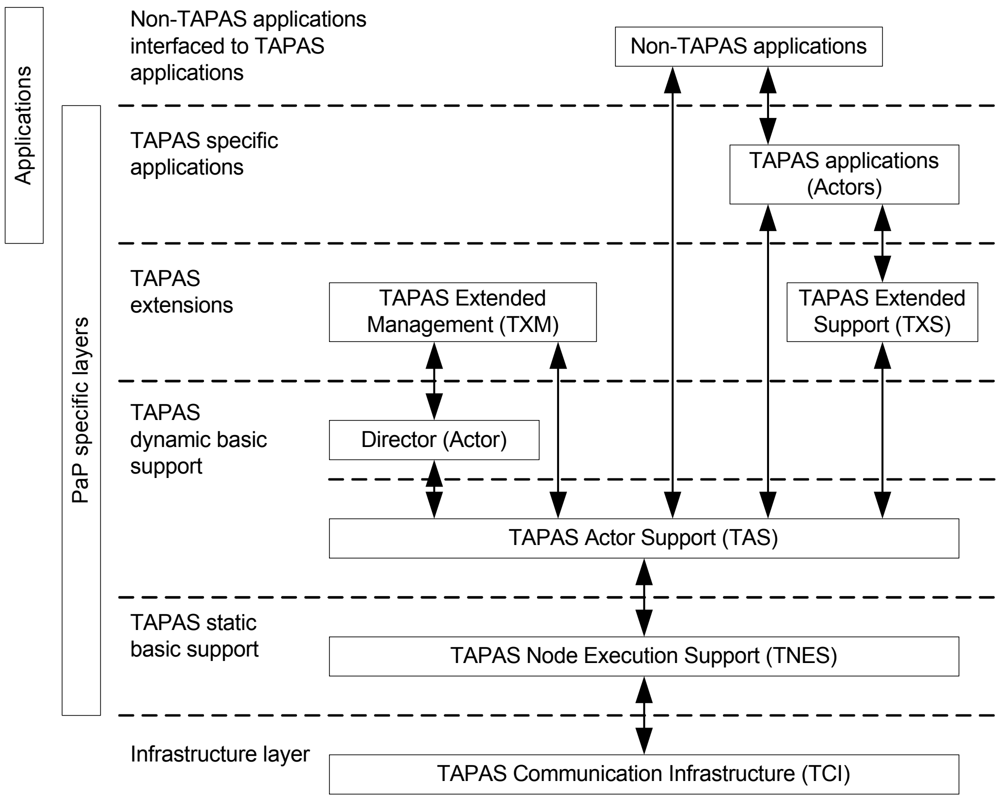

Applications
PaP specific layers
Non-TAPAS applications interfaced to TAPAS applications
Non-TAPAS applications
TAPAS specific applications
TAPAS applications (Actors)
TAPAS extensions
TAPAS Extended Management (TXM)
TAPAS Extended Support (TXS)
TAPAS dynamic basic support
Director (Actor)
TAPAS Actor Support (TAS)
TAPAS static basic support
TAPAS Node Execution Support (TNES)
Infrastructure layer
TAPAS Communication Infrastructure (TCI)

Figure 1-3: TAPAS layered design model – architecture

[2] Formelry known as PaP Node Exection Support (PNES), for the same reason as above.
[3] Formelry known as PaP Actor Support (PAS), for the same reason as above.

1.2 Introduction to MicroTAPAS

A specialized, downsized, version of the basic TAPAS architecture, MicroTAPAS, has been specified, and a prototype developed [LUHE], and is aimed at wireless devices with limited resources (i.e. memory, computing power and display), such as telephones and PDAs. The aim of MicroTAPAS is to develop a support architecture that, first of all, will enable these small devices to execute the TAPAS support system, and to provide functionality that will enable them to roam a TAPAS domain, making the mobility handling involved transparent to the end user. So far, the prototype of MicroTAPAS does not have any advanced support functionality for handling mobility, and is merely a version of TAPAS that enables the support architecture to execute on these resource-constrained devices.

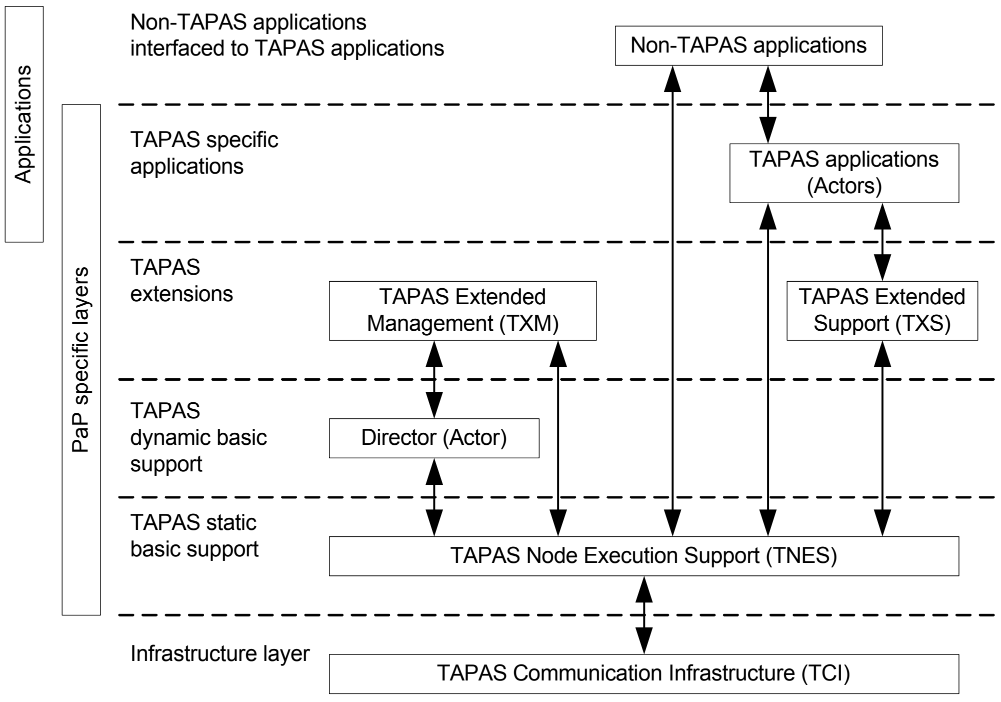

Applications
PaP specific layers
Non-TAPAS applications interfaced to TAPAS applications
Non-TAPAS applications
TAPAS specific applications
TAPAS applications (Actors)
TAPAS extensions
TAPAS Extended Management (TXM)
TAPAS Extended Support (TXS)
TAPAS dynamic basic support
Director (Actor)
TAPAS static basic support
TAPAS Node Execution Support (TNES)
Infrastructure layer
TAPAS Communication Infrastructure (TCI)

Figure 1-4: MicroTAPAS layered design model – architecture

Figure 1-4 [LUHE, page 9] illustrates the different layers of the MicroTAPAS architecture, and is a modified model of the architecture model shown in Figure 1-3 above. The MicroTAPAS architecture had to be simplified some, as the target devices could not easily handle the very extensive, but basic, architecture presented by TAPAS. Most notably is the merging of the TAS and TNES layers, as well as replacing Java/RMI communication with plain socket communication (not shown in figure), these changes are all explained in detail in [LUHE].

The basic network connectivity or radio access type applied and used in the prototype design and implementation was WLAN, which determined how the following mobility management mechanisms be worked out, i.e plain socket routines and pinging.

1.3 Introduction to mobility support

The first step in developing a truly mobile extension of TAPAS was to enable the architecture to be executed on small handheld devices, such as PDAs, and was realized with the introduction of MicroTAPAS. Although still a prototype, MicroTAPAS shows that TAPAS is well suited to run on these resource constrained devices, despite the fact that there are still a few issues related to the relative performance of MicroTAPAS on these devices compared to execution on laptop and desktop computers [LUHE, page 37]. The mobile nature of these devices makes them ideal for developing, and subsequent testing, of support functionality for handling various mobility issues, thus, MicroTAPAS will be used as the platform for designing and implementing these TAPAS extensions.

The task of this project is to design and implement TAPAS Extended Management (TXM) for providing extended mobility management routines and procedures, and TAPAS Extended Support (TXS) for providing all the needed behavior and performance extensions to the generic, but movable, actors, as well as mobility support for mobile nodes and terminals. Both of these extensions belong in the *TAPAS extensions* layer of Figure 1-4.

The TXM developed in this project contains routines and procedures that are beyond the functionality and scope of the Director object, which, in the MicroTAPAS case, will be realized by the MobilityManager (MM) object, likewise the TXS will contain routines and procedures that are beyond the functionality and scope of the MicroPNES object. The TXS functionality will be realized by the MobilityAgent (MA) object. Figure 1-5 illustrates where the TXM and TXS objects will be located on a running MicroTAPAS system. The figure is a modified version of the operating MicroTAPAS system example found in [LUHE, page 9]. The MobilityManager and MobilityAgent objects will be explained in greater detail in chapter 1.

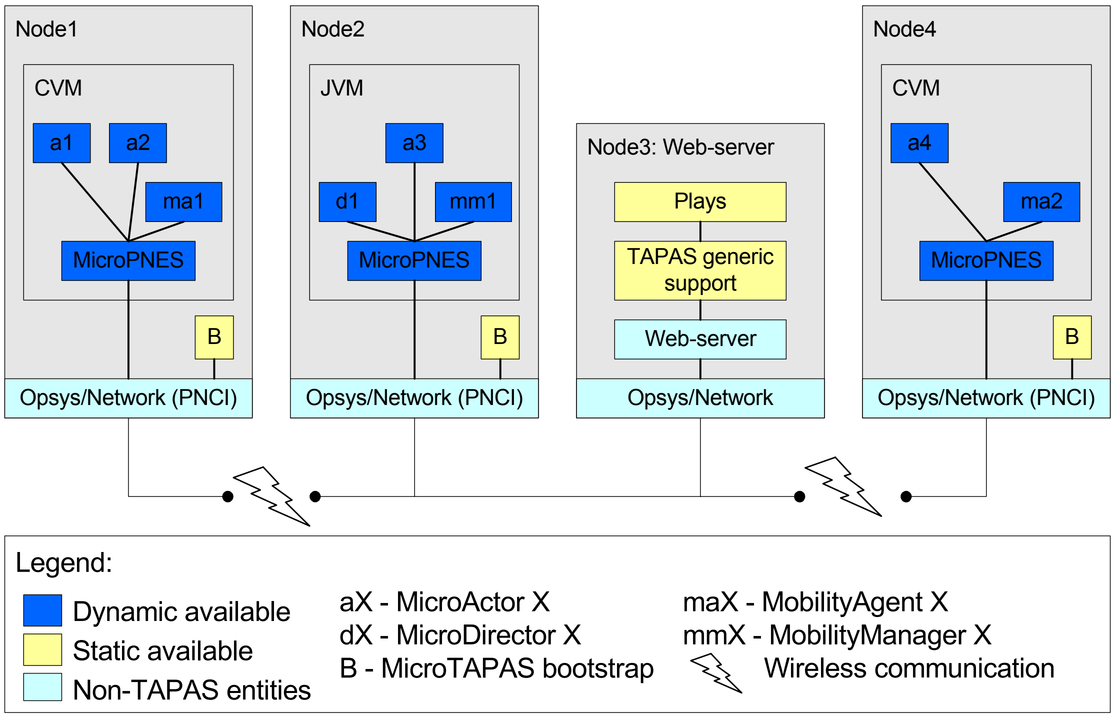


Figure 1-5: MicroTAPAS layered design model – operating MicroTAPAS system example (modified)

1.4 TAPAS papers on mobility and dynamic configuration

There are written a great number of papers, presentations and reports about the TAPAS architecture (37 at the time of writing), ranging from general architecture overview to specific implemented functionality. Most of this material is available on the Internet at [http://tapas.item.ntnu.no]. Three papers, nevertheless, builds the basis from which this project is the result, and a brief introduction to each of these is appropriate in order to provide the reader with a basic understanding of the main concepts behind the issue of mobility in TAPAS.

1.4.1 Mobility management in Plug-and-Play network architecture

[MALM1] presents mobility management within TAPAS, and the paper starts with a brief introduction to essential TAPAS concepts and the TAPAS layered model, before presenting different approaches for actor, terminal, user and session mobility management. For each type of mobility management, they have presented "an early set of mobility management algorithms or methods", as well as exploring "a few issues related to implementation design and propose as set of components to facilitate the deployment of this platform in the available PaP applications." The report concludes, in essence, that several issues discussed in the paper need further investigation. The authors argue that "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" and further "we need to find efficient ways to map our methods and procedures into proper applications and communication platforms".

1.4.2 User and session mobility in a Plug-and-Play architecture

"User and Session mobility in a Plug-and-Play architecture" [LILL], a master thesis, revolves around user and session mobility in TAPAS. The report first presents general object and engineering models for mobility support in TAPAS, before focusing on user and session mobility. In the chapter titled "The mobility architecture", the candidate states the functional and non-functional requirements for the framework and presents a UML use-case diagram of the user and session mobility, as well as class diagrams. In order to store user and session information between sessions, a system utilizing XML-files was devised. These XML-files store information such as roles, session descriptions and user profiles. The reminder of the thesis presents and discusses two sample applications, chat and file transfer, providing UML-diagrams and message sequence charts for both.

1.4.3 Mobility Support Framework in Adaptable Service Architecture

In [MALM4], a thoroughly presentation of a mobility support framework, aimed at a adaptable service architecture, is presented. The two first sections give the reader an introduction to TAPAS and mobility, and an overview of related work. The following six sections, which compromise the main content of the paper, deals with various aspects of mobility and, finally, a conclusion in section nine. The main section of the paper starts off with presenting an overall terminology framework and a section about service management and related considerations in the mobility handling architecture. Section five, six and seven each describes personal and user session mobility, user mobility and terminal mobility. Section eight presents some implementation issues and experiences.

2 Literature review

This literature review consists of four sections, where each section discusses a certain area with relevance to TAPAS. These areas are; active and controllable networks, mobile and cellular IP, mobile agents, multi agent and agent based systems, and finally support middleware and platforms. A quick overview will be given and the relevance to TAPAS discussed.

2.1 Active and controllable networks

Active Networks are classified by two approaches: active packets and active nodes. The first builds on the integration and deployment of services in the user flow, while the second is based on deploying services dynamically in nodes.

The U.S. Department of Defence's (DoD) Defence Advanced Research Projects Agency (DARPA) has set in motion an active networks program that aims at producing a new networking platform. The architecture "is based on a highly dynamic runtime environment that supports a finely tuned degree of control over network services. The packet itself is the basis for describing, provisioning, or tailoring resources to achieve the delivery and management requirements." [DARP]

The Distributed Computing and Communications Lab at Columbia University have developed a programming language and environment, called NetScript, for building networked systems, and is thoroughly described in [COLU]. The programs that are designed and implemented in this language are organized as mobile agents that after deployment to remote systems can be executed either under local or remote control. The purpose of the project is to simplify the development of networked systems, and their remote programming, as "networked systems are difficult to design, implement, deploy and manage." [COLU]

An Active Networks project at Massachusetts Institute of Technology (MIT), which has been funded by DARPA, has developed the Java-based Active Node Transfer System (ANTS), for experimenting with active networks [MITE].
There are a number of other projects on active networks as well, among them the SwitchWare project undertaken by the University of Pennsylvania and Bellcore, which is described in [UPEN].

These are just a few of the research projects going on in the field of active networks. There are some obvious similarities between TAPAS and the technologies presented above, in particular the 'active nodes' approach. However, TAPAS is based on code-on-demand and not pre-programmed packets, as is the case for active networks. The nodes in TAPAS need only to run a fixed-sized executable and have a set of basic settings, such as initial web address and configuration files, while active networks need to include how packets be interpreted in the packets themselves.

2.2 Mobile IP and Cellular IP

When a computer is connected to a specific network, it is allocated an IP address, when that computer moves to another network it is given a new IP address, i.e. by Dynamic Host Configuration Protocol (DHCP). This scheme works fine in most cases, but a problem arises if files or resources on that computer are sought by others, since they would not know that computers address.

This is where mobile IP comes into the picture, and with this transparent scheme, computing continues as normal when a host is moved from one subnet to another. When the computer is connected to its home base, packets are routed in the usual way. When it is connected elsewhere, two agent processes take over the routing, the home agent (HA) and the foreign agent (FA) running at fixed nodes on the two subnets. When the mobile host leaves its home domain, the HA is informed of this, and the FA of the visited domain relays back to the HA that the host is available in that domain. The HA then operates as a proxy, relaying all traffic to the mobile host through the visited domains FA.

According to [COUG, page 104], “The MobileIP solution is effective but hardly efficient. A solution that treats mobile hosts as first-class citizens would be preferable, allowing them to wander without giving prior notice and routing packets to them without any tunneling or rerouting.” This is clearly a drawback to the technology, but could be amended in the future to work along the lines of how cellular phones roam networks.

IP Mobility is also described at great depth in several Request For Comments (RFC) documents, among these are “IP Mobility Support” [RFC2002] and “IP Mobility Support for IPv4” [RFC3344]. These two RFC’s cover very specific mechanisms dealing with issues related to IPv4.

There is no support for terminal mobility in TAPAS, but, as explained in section 1.4, [MALM1] discusses a Mobility Management platform for TAPAS. In addition [MALM1] and [LILL] have proposed a scheme for user and session mobility, using the principles of mobile IP, including home and foreign agents.

2.3 Mobile agents, multi agent and agent based systems

A mobile agent is a program, script or package that physically travels around a network, and performs operations on hosts that have agent capabilities. These agents, which operate autonomously, usually has very specific tasks, such as fetching prices of merchandise from on-line stores, or to collect weather information. Apart from interacting with all sorts of operating systems, databases or information systems, agents can also interact with other agents, meeting in agent-gathering places to exchange information. There are a number of different mobile agent architectures and languages available today, such as Knowledge Query and Manipulation Language (KQML) as presented in [UMBC], which is part of the broader ARPA Knowledge Sharing Effort [STAU], and is a “language and protocol for exchanging information and knowledge”. Although agent technologies have received a lot of attention in recent years, [REID] argues that “mobile agency has failed to become a sweeping force of change, and now faces competition in the form of message passing and remote procedure call (RPC) technologies”.

The very autonomous nature of mobile agents sets them wide apart from the basic TAPAS’ request/response interactions, although the resulting action might be comparatively equal, and a TAPAS node can almost be seen upon as a stationary agent. However, with the introduction of ActorMobility concepts into the TAPAS architecture presents an Actor model that can almost behave as a mobile agent. A MobileActor is a controllable mobile agent with limited autonomousity.

2.4 Support middleware and platforms

Two available middleware platforms are discussed in this section, CORBA and Jini.

2.4.1 CORBA

The Common Object Request Broker Architecture (CORBA), by the Object Management Group (OMG) [OMG1], is an open and vendor independent architecture and infrastructure that computer applications can use to work together over networks. The architecture uses a standard protocol, IIOP, whereby a CORBA-based program "from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network." [OMG2].

CORBA is, thus, only an architecture and infrastructure which applications can use to communicate and interoperate over a network, a networking technology. TAPAS on the other hand, is a software architecture that facilitates for dynamic introduction of new distributed services, or upgrading of existing ones, in a communication network. TAPAS could hence have been built using the principles of CORBA to carry out communication, and as such is not a 'competing' technology in that respect.

2.4.2 Jini Network Technology

"Jini network technology (which includes JavaSpaces Technology) is an open architecture that enables developers to create network-centric services – whether implemented in hardware or software – that are highly adaptive to change. Jini technology can be used to build adaptive networks that are scalable, evolvable and flexible as typically required in dynamic computing environments." [SUN1]

By using objects that move around the network, the Jini architecture makes each service, as well as the entire network of services, adaptable to changes in the network. The Jini architecture specifies a way for clients and services to find each other on the network and to work together to get a task accomplished. Service providers supply clients with portable Java technology-based objects that give the client access to the service. This network interaction can use any type of networking technology such as RMI, CORBA, or SOAP, because the client only sees the Java technology-based object provided by the service and, subsequently, all network communication is confined to that Java object and the service from whence it came.

Jini offers some of the same promises as TAPAS, but its foundations are built on some of the same principles as movable agents and its autonomous nature is thus not comparable to the explicit request/response interactions of TAPAS.

3 Mobility support for wireless devices

This chapter constitutes the first of the two main chapters of this thesis, the second one being chapter 4; Implementation, and contains three sub-sections, in addition to this introduction. The first section introduces the mobility framework. This framework is the basic building block of the mobility extension to TAPAS. The second and third sections discuss terminal and actor mobility, respectively, and how these can be achieved.

The aim of this thesis, as stated in section 1.3 on page 1, is to design and implement TAPAS Extended Management (TXM) for providing extended mobility management routines and procedures, and TAPAS Extended Support (TXS) for providing all the needed behavior and performance extensions to the generic, but movable, actors, as well as mobility support for mobile nodes and terminals. The specific task is thus to design and implement actor and terminal mobility in a TAPAS system, realized by MobilityManager (MM) and MobilityAgent (MA) objects. These two TAPAS extension objects may collectively be referenced to as MobilityEnities. An example TAPAS system deployment might look like the one illustrated by Figure 3-1 below. This system consists of three domains, A through C, and domain B is divided into two sub-domains, B1 and B2. All domains consists of at leas one Dir/MM node, and any number of MM and/or MA nodes. The specific distribution of Dir/MM, MM and MA nodes will be explained in section 3.2.

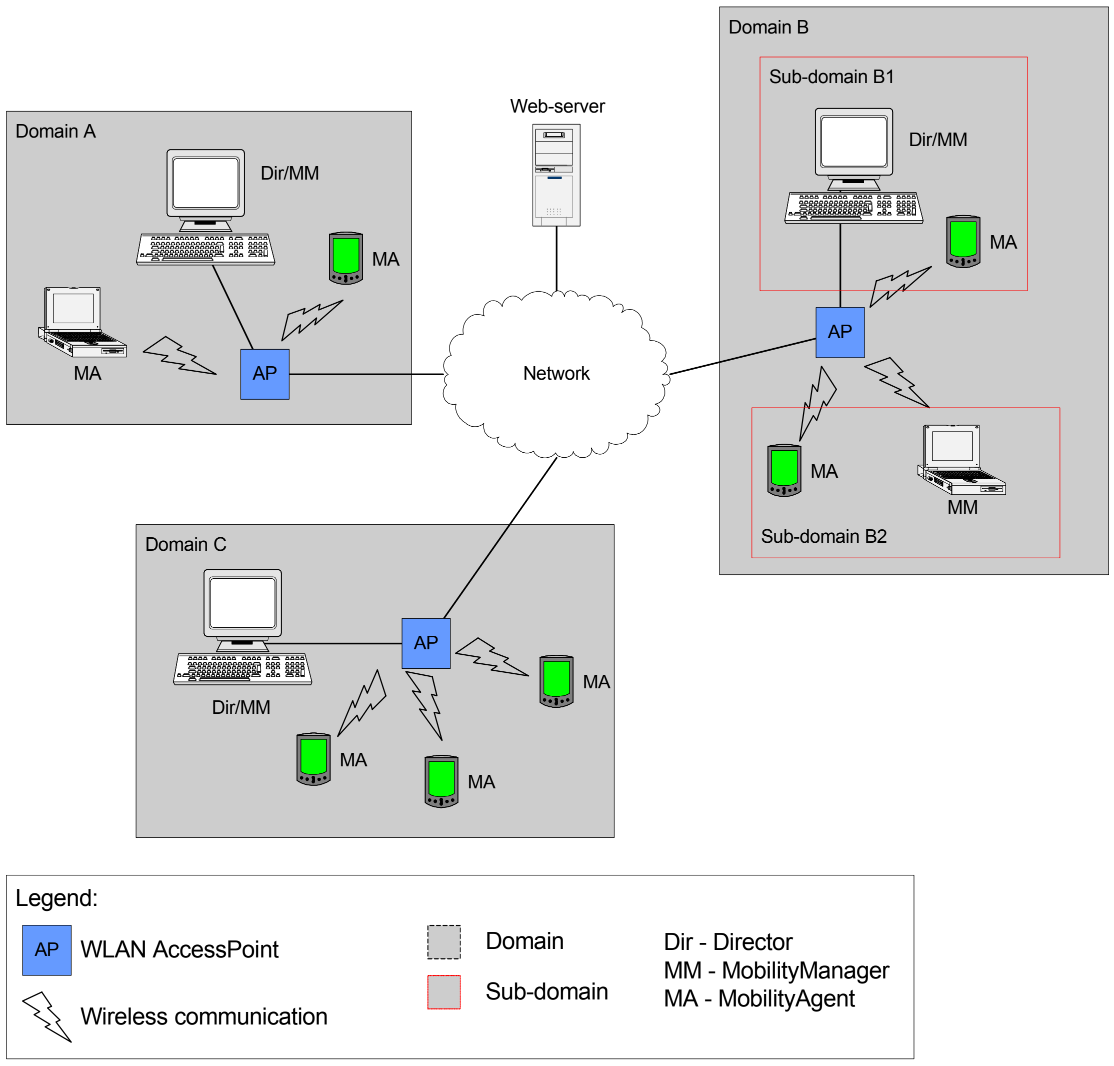


Figure 3-1: Example MicroTAPAS system set-up

The goal, therefore, is to design management and support functions that; a) will allow a terminal to move from any one domain to another, including movement between two sub-domains, and b) to allow an actor located on any of these nodes to move from any one node to another (or within the same node for that matter). The system should at all times be kept in a consistent state, and any registries that hold information must therefore constantly be kept up-to-date.

3.1 Definitions

This is a list of definitions commonly referred to throughout the rest of this report:

- Node is a physical network entity capable of taking part in TAPAS-based services, by running TAPAS support and TAPAS service component(s). This may be directly mapped to PCs, handhelds, mobile phones, or any other device with a computing capacity and operating memory capable of running external applications. A node is uniquely identified by its location. Terminal is one type of node that is associated with end users as their means of accessing services.

- Location (Access point) is the physical address information. This can be network address, geographical location, etc. A location is used to uniquely address nodes running TAPAS service components.

- Domain represents a population of actors and/or nodes managed by one director. Domain concept in TAPAS is used to manage and administrate the federation of responsibility between different director objects. In TAPAS two types of domains are distinguished: Home domain and Visitor domain.

- Sub-domain is a collection of one or more nodes in a domain that share their director with a collection of one or more other nodes, but have their own instance of a MobilityManager.

- Actor is the generic object of TAPAS with a generic behavior, which can behave according to a manuscript specifying certain functionality.

- Actor child session represents a session initiated and maintained by an actor, which results in instantiating new actors with their respective data, role-sessions, settings, etc.

- Role-Session is a projection of the behavior of an actor with respect to one of its interacting actors. It represents a relationship between two actors (initiator and cooperator).

- Actor Mobility stands for the movement of instantiated functionality at a node that is executed by an actor. This implies a change and update of the actor location-specific information.

- Role-Session Mobility stands for the re-instantiation of role-sessions of moved actors by re-creating them at the new location where the moved actors is re-instantiated.

- Terminal Mobility is the movement of terminals and changes their location while maintaining access to services and applications.

- Mobility Agent is the component of the architecture that is responsible for managing terminals location-related information. It performs location updates when a terminal changes location.

- Mobility Manager is responsible for managing actors and terminals connectivity and mobility.

Figure 3-2 shows the object model of the TAPAS Mobility Platform, and is a revised version of Figure 1-1. Those objects that have been introduced since the earlier version are shaded grey.

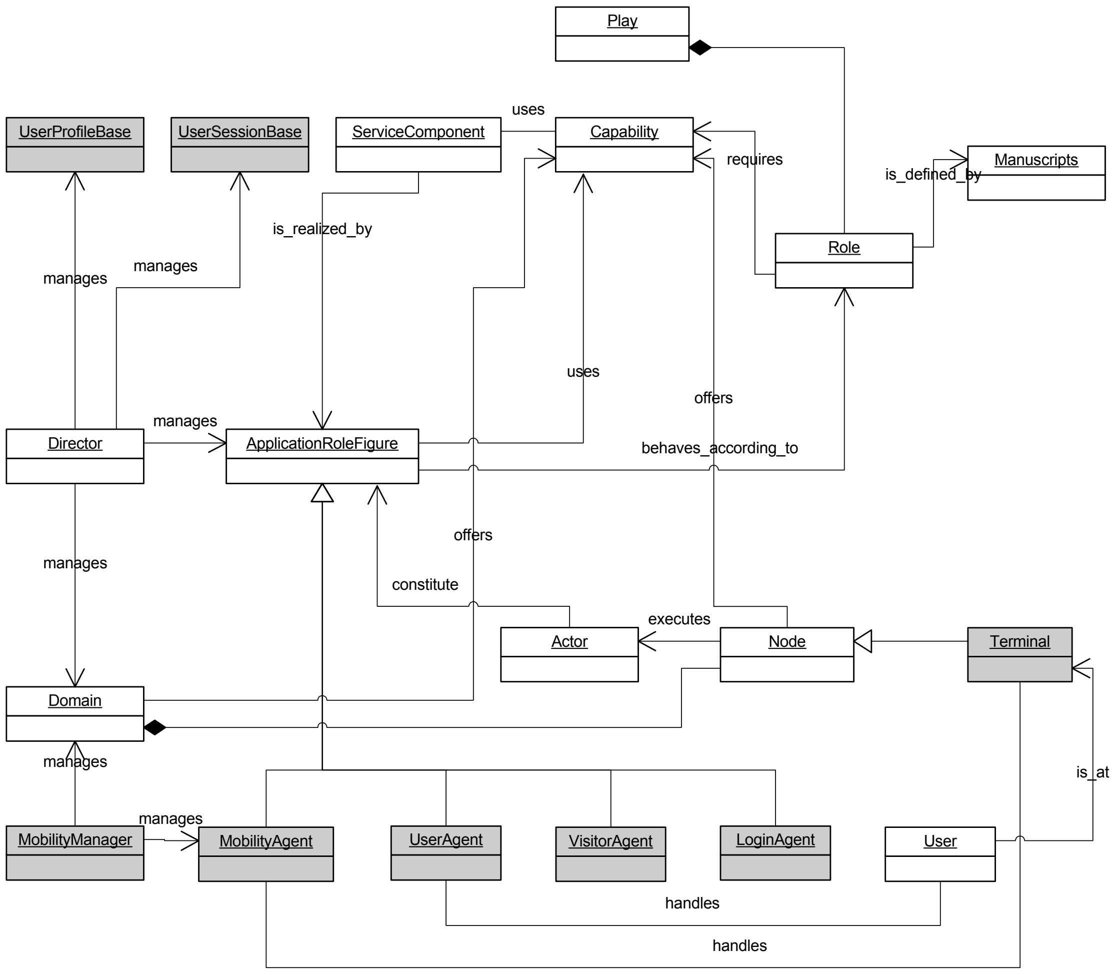


Figure 3-2: TAPAS Mobility Platform (Object Model)

3.2 Mobility framework

The MicroTAPAS mobility framework lies at the hart of the TAPAS extended mobility platform, and provides internal support functionality for the mobility entities that are responsible for handling the various mobility issues. The framework, offering internal support functionality only, does not deal directly with such issues as terminal or actor movement, but provides the necessary functionality for storing, accessing and updating various types of information, routing of requests to and from actors, and offers a common interface for application development. Figure 3-3 illustrates how the mobility framework fits in with the TAPAS extensions layer, and its TXM and TXS objects.

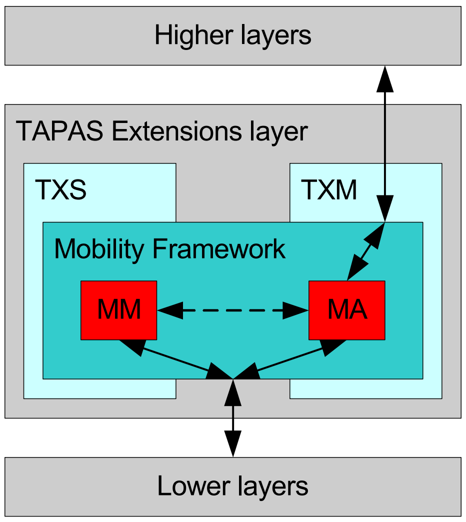


Figure 3-3: Mobility framework

3.2.1 Distribution of mobility entities

This section takes a look at where the two mobility entities are to be placed in a TAPAS domain, i.e. the hierarchy of the various entities and how they relate to each other. Figure 3-4 illustrates how the hierarchy of the different entities will be ordered in one particular node consisting of two sub-domains.

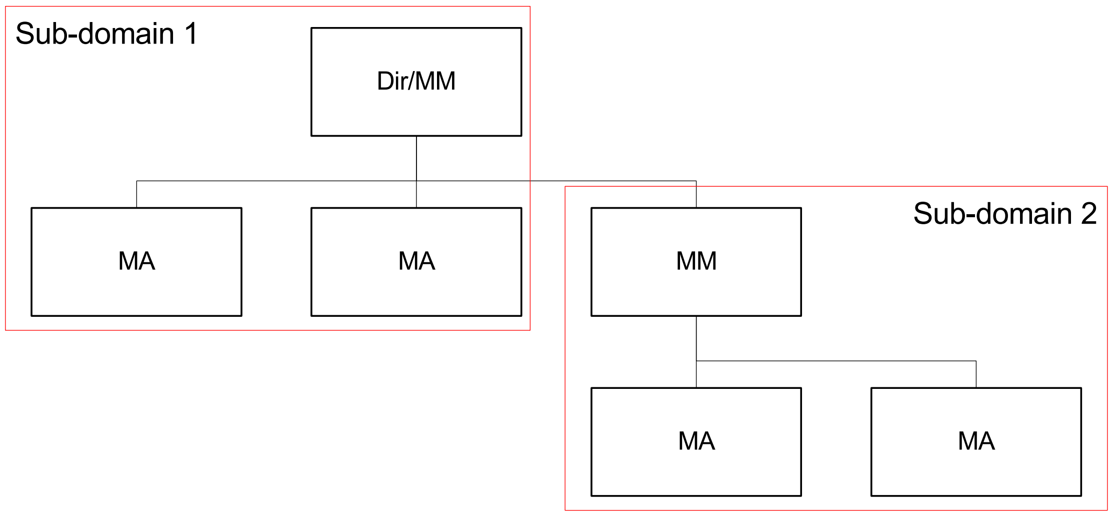


Figure 3-4: Hierarchy of mobility/TAPAS entities

As can be seen in Figure 3-4, it is possible to order the different MM nodes in a multi-level hierarchy. This feature will make it more resource efficient to setup and manage larger TAPAS networks, as the management traffic overhead will not increase exponentially with respect to the number of nodes/domains in a network. With multilevel hierarchies, management messages are only transmitted between the affected parties. If one had a completely flat structure, all management messaging would have to be transmitted to all the connected MM's in the network.

3.2.1.1 Director and MobilityManager nodes

A Director node is a node where the director of the TAPAS domain runs, while any other node, client nodes for instance, are nodes that do not run a director. In any node that runs TAPAS support actor instances can be instantiated. TAPAS domain contains in addition to director node and client nodes a web server to serve as the play repertoire.

A traditional TAPAS domain contains one Director node, and any number of ‘client’ nodes (i.e. nodes that require the services of a Director). With the introduction of mobility entities, the notion of only Director and client nodes had to be expanded. On each node running a Director, a MobilityManager is also present, and that type of node is called a Dir/MM node, as seen in the top position in Figure 3-4. The reason for co-locating the Director and MM is simply the fact that to successfully provide mobility to all nodes and actors, each domain must have a MM to manage the mobility functionality, and as each domain already contains a Director node, it was natural to co-locate the two at a node.

3.2.1.2 MobilityManager nodes

There exists no definition of sub-domains in the traditional TAPAS architecture, but the addition of these semi-autonomous domains was deemed necessary in order to increase the system flexibility. A sub-domain consists of one MM and any number of ‘client’ nodes, and share its Director with all the sub-domains located within the same super-domain. In Figure 3-4 there are two sub-domains, shown by the stippled read box. If one considers a domain with many connected terminals, and some of these terminals are constantly roaming the network (or its actors are), the MM of that domain would constantly be engaged in updating registries and moving terminals and actors between domains and sub-domains, and its workload would certainly outweigh the workload of the Director. By allowing more than one MM to share a Director one can distribute the work done my these managers over several terminals, and increase or decrease their numbers as necessary, without having to update each terminal with a new set of configuration files, as each MM is assigned a range of addresses considered its own and will automatically take charge of those terminals its supposed to be manager for.

3.2.1.3 MobilityAgent nodes

Each node in a TAPAS domain which is neither a Dir/MM node nor MM node contains a MobilityAgent, and as such is considered a ‘client’ node. The MA will be responsible for sending updates to its local MM, i.e. synchronizing, and will be the mobility interface for actors upwards, and the terminal downwards. All mobility requests to and from the MM are routed through the local MA.

3.2.2 Actor model

Before moving on to discuss the particulars of terminal and actor mobility, a basis for an extended actor model needs to be established, due to the limitations put forward by the existing, basic, actor model. A short introduction to the basic actor model is given, before a more detailed explanation of the extended version is presented.

3.2.2.1 Basic actor model

In the general actor model, there is a distinction between two main functionality constituents, support functionality and behaviour. Functionality is utilised by methods, and therefore will be denoted as Methods, and a manuscript is equivalent to a behaviour definition, and therefore will be denoted by Behaviour.

Support functionality is mainly predefined routines and procedures with a well-established definition, access means, and mechanisms. This is a property of the generic actor object, which is part of the TAPAS platform or middleware, and application functionality or the manuscript that contains the definition of the behaviour of the application actor. This is a property of the application role-figure object (Actor + manuscript).

Behaviour, on the other hand, will be characterised by a specification that obeys the rules of a state machine model, and will have a current state as a description of its status.

In Figure 3-5 there is a basic sketch of the actor model with methods and behaviour to stand for these two parts respectively. A method consists of a set of instructions and actions to be executed within a context that works mainly as a method call with calling and return procedures. In the figure, a distinction between movable and un-movable components has been made.

Dynamic and static components, in this context, simply highlights that some components are changed/altered during normal operation of the actor. For example, an actor's state is changed *dynamically* depending on its state (i.e. the actor is ready or busy). The behaviour, in contrast, is *static*, in the sense that for it to change a new manuscript must be downloaded and instantiated.

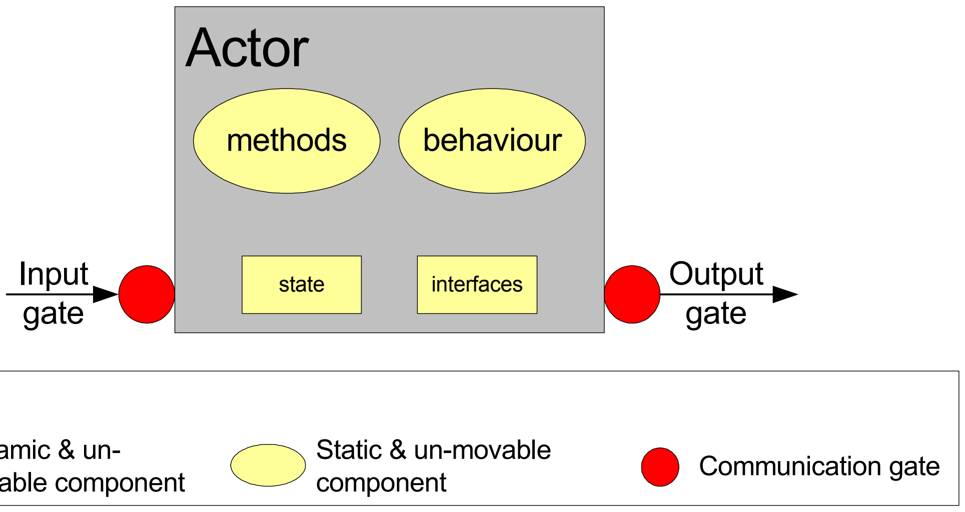


Figure 3-5: TAPAS basic actor model

This distinction between methods and behaviour gives the model more flexibility and maps it properly to existing concepts in middleware platforms and programming paradigms, as well as fitting the specification techniques applied in teleservices.

3.2.2.2 Extended actor model

The basic actor model provides a well defined definition of an actor to be used in a static environment (i.e. the actors or terminals are not moved). With the introduction of Actor and Terminal mobility, however, a slightly more complex structure is needed in order to comply with the specification. When an actor or terminal is moved, the actors affected by the operation needs to be able to continue their operation after being moved – something the basic actor model could not handle. An illustration of the TAPAS extended actor model is shown in Figure 3-6.

Movable components are those components that are *moved* (copied) to a new instance of the actor (as part of the actor move procedure). Un-movable components, on the other hand, are not moved but *re-instantiated* at the new location (i.e. downloaded from the codebase).

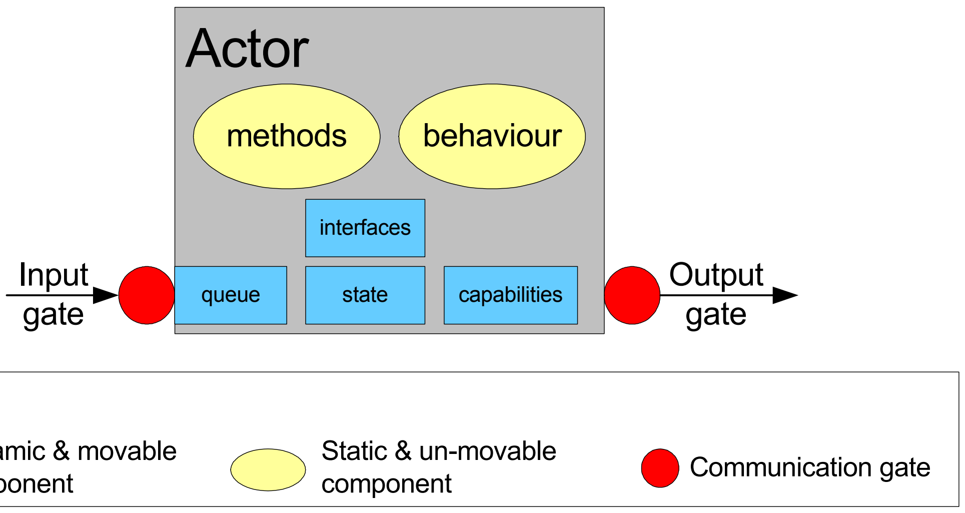


Figure 3-6: TAPAS extended actor model

In the four subsections presented below, four properties of the extended actor model is presented and explained; interfaces and role-sessions, behaviour definition, capabilities and requirements, and queued requests

3.2.2.2.1 Interfaces and role-sessions

Some platforms often use the term *interface* to describe the relationship between entities in the platform. In the TAPAS architecture, the term *role-session* is used to describe such relationships instead of interface. All application-actors in the TAPAS architecture have at least one role-session (refer to Figure 3-2) associated with them – its creation, or initial, role-session. This role-session is a relationship between the creator of the actor, referred to as the initiator, and the created actor itself, referred to as the co-operator. The co-operator can at a later stage become the initiator of one or more *other* actors, and in the end one can have a whole tree of relationships. Figure 3-7 illustrates how such a tree might look like. Each left-hand role-session (RS) is the actor's co-operator role-sessions, and the right-hand ones are its initiator role-sessions.

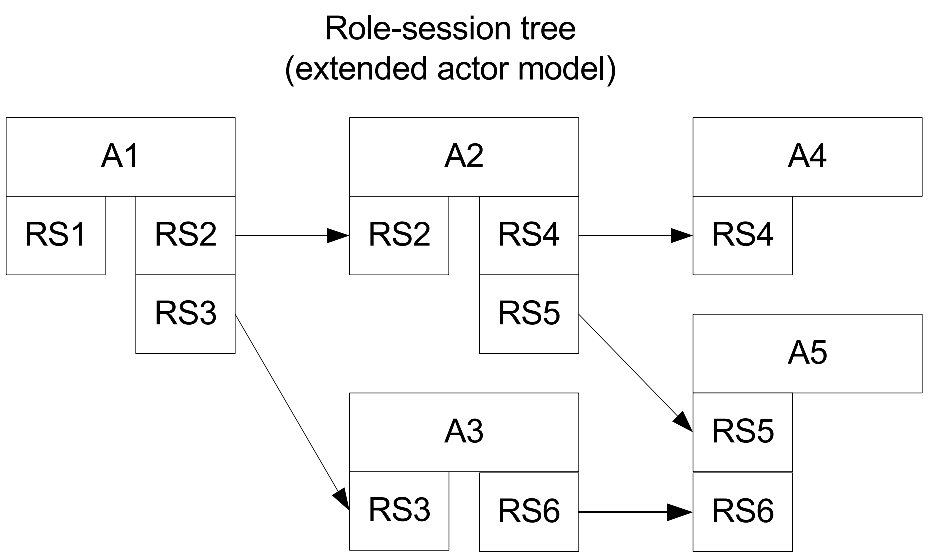


Figure 3-7: Role-session relationship-tree

The first application-actor in any TAPAS system will be created with a TNES object as its initiator (the TNES object is not shown in the figure). Each instantiated actor can have more than one initiator, as is the case for actor A5 in Figure 3-7, and can be the initiator of zero or more other actors. Each role-session, however, can only represent a one-to-one relationship between one initiator and one co-operator.

When an actor is moved (or a terminal with one or more instantiated actors), all the role-sessions the actor is either the initiator or co-operator of must be updated in such a way that the system is not left in an inconsistent state.

3.2.2.2.2 Behaviour definition

In the actor model there is a distinction between two main functionality constituents. The first is the support functionality which consists mainly of predefined routines and procedures with a well-established definition, access means, and mechanisms. This is a property of the generic actor object, which is part of the TAPAS platform or middleware and application functionality or the manuscript that contains the definition of the behaviour of the application actor. This is a property of the application role-figure object (Actor + manuscript). Secondly, behaviour will be characterized by a specification that obeys the rules of a state machine model, and will have a current state as a description of its status. This distinction between methods and behaviour gives the model more flexibility and maps it properly to existing concepts in middleware platforms and programming paradigms, as well as fitting the specification techniques applied in tele-services.

3.2.2.2.3 Capabilities and requirements

Some actors in a TAPAS system might have special requirements that must be fulfilled by the target node for the actor to successfully carry out its tasks. It would, for example, be essential for an actor which acts as a printing proxy that the node on which it is plugged in has an attached printer. Another example would be an actor for displaying colour pictures – it would require that the node on which it is plugged in has a display that could handle, say, 65K colours. An actor's *required capabilities* should therefore be matched with the target node's *offered capabilities* prior to the actor being plugged in at that node.

[AAGF2] proposes an extensive framework for capability specification and selection, including a very detailed and broad example [AAGF2, page 4] which includes a range of offered and required capabilities. The incorporation of such an extensive framework is far beyond the scope of this thesis, but a limited framework is included to enable the actors to exercise some control on the type of executing environment they are about to be plugged into. The capabilities that can be consumed for this prototype of mobility support are processing power, operating memory, screen resolution and number of colours on the display.

These capabilities are defined as follows:

- *Processing power* is the clock frequency of the processor of the device, and will be measured in megahertz (MHz).
- *Operating memory* refers to the total amount of memory that the device has for executing applications. This quantity will be measured in megabytes (MB).
- *Screen resolution* is the maximum resolution of the display of the device, both horizontally and vertically, and is measured in pixels.
- *Number of colours* is the total number of colours that the display may draw simultaneously at any given moment. A black and white display, for example, has only two colours, while a modern PDA might be able to draw as many as 65'000 colours simultaneously.

The actor's required capabilities should be stored in a file that is accessible by the TAPAS system, either locally or at well known location. These capabilities are static and should not be changed over the lifetime of the actor. The node's offered capabilities should, if possible, automatically be determined by the TAPAS system at start-up, and continuously monitored throughout the lifetime, or existence, of the node. Those capabilities that can not be determined automatically should be stored in a file in the local file system, at a location that is known to TAPAS.

When a node receives a request from the director to plug-in an actor, actorPlugInReq (APIR), the TNES instance at that node should locate the actors required capabilities and compare them with the capabilities that the node has to offer. The APIR can be either a stand-alone plug-in request, or a plug-in request as part of an actor move procedure. If all the offered capabilities are equal to, or exceeds the required capabilities, the plug in of the actor is given a green light. Otherwise, the director is informed of the discrepancy and the plug in is a failure.

In this version of mobility support, actors can only consume capabilities, and not offer any to other actors or nodes. In a future version of this mobility extension, where a more complete framework for capability specification and selection is to be incorporated, actors should also be able to offer capabilities.

3.2.2.2.4 Queued requests

The movement of an actor, either within a node, or between two nodes, is not an instantaneous event, and, theoretically, the actor could receive requests during the move operation. The extended actor model should implement a scheme that takes this into consideration. One possible solution, and the simplest, is to simply discard all the requests. This might be a viable option for some applications; especially for those where the old requests are out-dated by the time the new actor is ready to consume them. An example would be an application for streaming sound over the network, where it makes no sense in replaying sound-bits that might be several seconds old. This can be compared to using UDP datagrams for streaming speech in IP telephony over a network.

Another possible solution is to queue all incoming requests during the movement phase. All requests to the old and soon-to-be-discarded actor and to the new and not-yet-operational actor would be put in queues. When the new instance of the actor is initialized and ready to resume operation, all the queued request will be executed in a systematic manner. In a chat application, for example, this scheme would offer a better solution than to throw away all incoming messages while the actor is relocated.

Realizing that different actors might want to use different schemes for handling in-between-requests (requests received while in a movement phase), the most flexible solution would be for the actors themselves to choose this at runtime. This could be realized by including the desired option in a text file that could be read by the system if the actor was about to be moved from one location to another.

3.3 Terminal Mobility

In TAPAS, terminals realize the interface towards the end user, whilst nodes are viewed fixed as seen from their location point of view, though they might be given changeable or dynamic network addresses. The mobility as a feature is mainly provided for terminals, as end users want to access their subscribed services whilst on the move at different locations.

3.3.1 Definition

A terminal, as defined in section 3.1, is one type of node that is associated with end users as their means of accessing services. The reader should keep in mind that the term terminal is replacing more concrete terms as PDA or laptop. Thus, a reference to a terminals address would be the same as referring to the terminals IP-address.

A terminal is said to move when its physical network address is changed [MALM1, page 11]. In general, there are two ways in which a terminal/node can have its address changed; either by changing the network properties/configuration of the node (explicit), or the node is given a new network address by a central network element (implicit), i.e. by dynamic host configuration protocol (DHCP).

Terminal mobility will enable a terminal to move between different (sub-) domains, while the terminal’s user can access his or her subscribed services, with none or minimal disruption.

3.3.2 Overview

In order to achieve terminal mobility one needs to track the movement of the terminals participating in a TAPAS service network. Thus, a manager should be responsible for updating the location of all nodes that participate in a possible TAPAS service. This central and supervisory agent will be referred to as MobilityManager, and runs at an address known to all other nodes, for instance its network location may be part of a configuration file. MobilityAgent will issue LocationUpdate procedure, upon changing terminals location, and NodeDiscovery procedures, once a communication is required with other terminals or nodes.

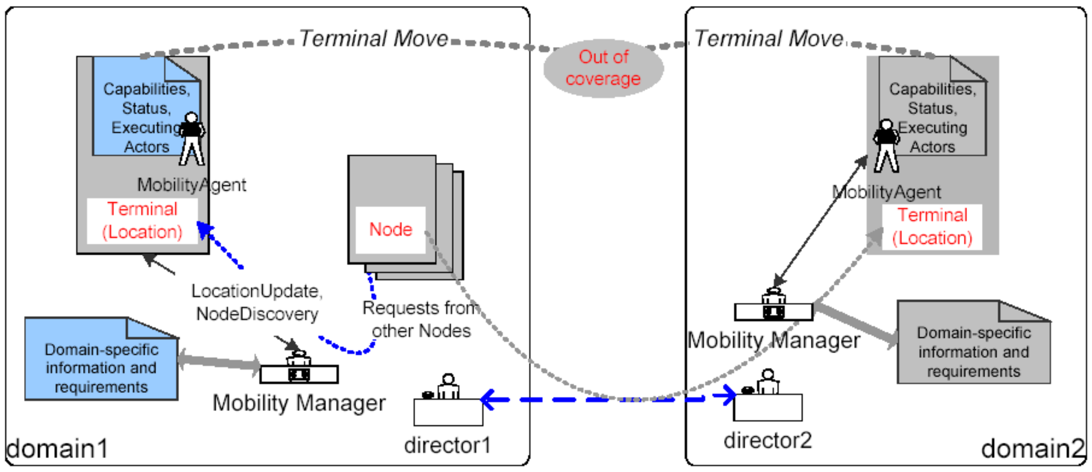

Terminal Move
Out of coverage
Terminal Move
Capabilities, Status, Executing Actors
MobilityAgent
Terminal (Location)
Node
LocationUpdate, NodeDiscovery
Requests from other Nodes
Domain-specific information and requirements
Mobility Manager
director1
domain1
Capabilities, Status, Executing Actors
MobilityAgent
Terminal (Location)
Mobility Manager
Domain-specific information and requirements
director2
domain2

Figure 3-8: Terminal mobility in TAPAS

Figure 3-8 [MALM4, page 14] demonstrates a general case of terminal mobility. A terminal moves from one domain to another, from domain1 to domain2, while its MobilityAgent ensures that MobilityManager is updated on this movement. However, when it reaches a limit of one domain, or the so-called out-of-coverage, it is considered as inaccessible. Meanwhile, requests from other nodes addressed to this terminal should be preceded by a NodeDiscovery procedure, which is executed through the corresponding MobilityManager is a domain. Upon entering another domain a terminal may be allowed to access certain services based on a director-to-director authentication process. MobilityManagers operate according to a set of domain specific set of setting and requirements, which govern the privileges and access rights specific users or terminals might have.

3.3.3 Terminal mobility scenarios

There are three different scenarios which the terminal mobility support should be able to handle. These scenarios, of different complexity, deals with which controlling, or supervisory, entities are switched, if any. The controlling entities, MobilityManager and Director, have well defined areas of operation (domains), i.e. they control all terminals within a certain address range. This was described in section 3.2.1. This section thus deals with the different scenarios occurring when a terminal moves within one domain or between different domains. The three scenarios are described below.

3.3.3.1 Within same sub-domain

The first and most simple, terminal move operation occur when a terminal moves within a domain or sub-domain. This means that the terminal will have the same responsible MobilityManager and Director as it did prior to the move.

3.3.3.2 Between two sub-domains within the same domain

The second scenario is when a terminal moves between two sub-domains within the same super domain. During such an operation, the terminal's responsible MobilityManager is switched, but the Director will be the same.

3.3.3.3 Between two domains

The third, and most complex, operation occurs when a terminal moves between two separate domains. This operation requires that both the controlling MobilityManager and Director entities are exchanged for those which govern the new domain.

3.4 Actor Mobility

3.4.1 Definition

Actor mobility is defined as the movement of instantiated functionality from one node, along with its properties, such as behaviour, capabilities, role-sessions, to another node, in a transparent manner for all other actors. Actor movement can be initiated as a result of different reasons, e.g. changed capability requirements, deterioration of available resources, dynamic change in configuration, change in functionality, or as a consequence of terminal mobility. Moved actors need to be able to carry on their functionality after being re-instantiated at the new location.

3.4.2 Overview

An actor in the TAPAS context, as presented in section 3.2.2, is described by the following parts: a) set of interfaces or role-sessions, b) behaviour definition that has a state, c) set of capabilities and requirements, and d) queue of incoming requests. Mobility in this context can be achieved by re-instantiation of actors with these parts preserved (if desirable). Different strategies might be employed to ensure the successful movement of an actor, for example, how to handle the queue of incoming requests; should it be dismissed, transferred to the new instance or be carried out prior to moving the actor.

In essence, the move procedure is defined by, or equivalent to, a sequence of ActorPlugOut, ActorPlugIn, CapabilityChange, CreateInterface, and BehaviourChange requests, which are part of the basic architecture, and used to destroy an actor, instantiate it, update its capabilities, set a role-session with another actor, and change the manuscripts it executes, respectively.

To allow for different interpretations by run environments, programming languages, and operating aspects, certain conditions must be specified that will control this procedure. A basic set of conditions might be; a) capability and interface parts may be reconstructed through applying supplementary CapabilityChange and CreateInterface procedures, b) behaviour part must be the same, and state information may be transferred using BehaviourChange, and c) queue and method parts will be dismissed.

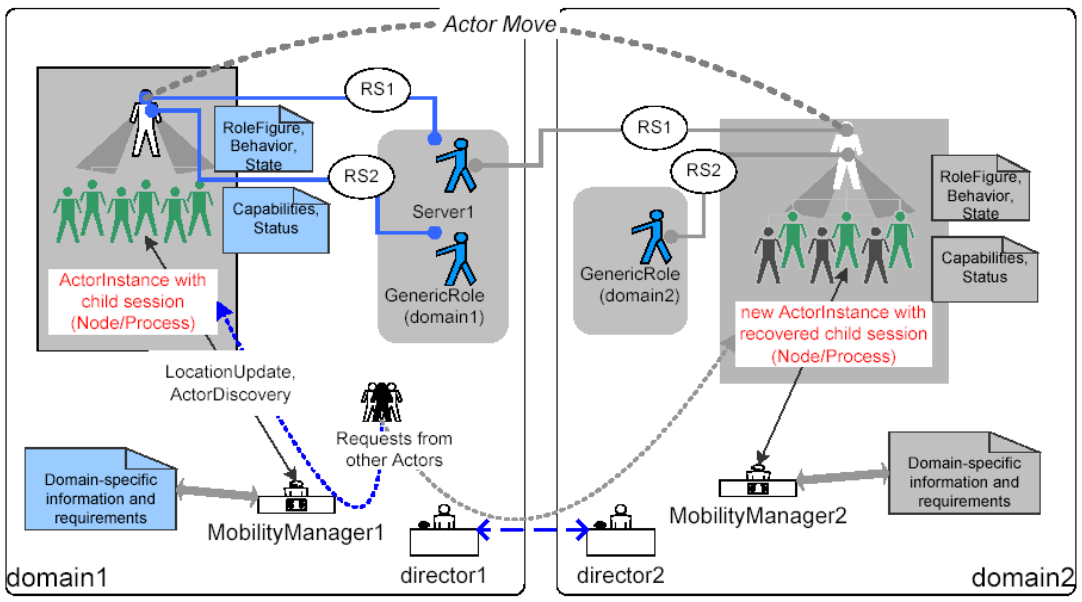

Actor Move
RS1
RS2
RoleFigure, Behavior, State
Capabilities, Status
Server1
GenericRole (domain1)
ActorInstance with child session (Node/Process)
LocationUpdate, ActorDiscovery
Requests from other Actors
Domain-specific information and requirements
MobilityManager1
director1
domain1
RS1
RS2
GenericRole (domain2)
RoleFigure, Behavior, State
Capabilities, Status
new ActorInstance with recovered child session (Node/Process)
Domain-specific information and requirements
MobilityManager2
director2
domain2

Figure 3-9: Actor mobility in TAPAS

Figure 3-9 [MALM4, page 12] presents a general scenario for actor mobility which involves two different domains, *domain1* and *domain2* and a general actor instance. In accordance with a set of conditions, i.e. ‘a’, ‘b’ and ‘c’ presented above, the actor, which might encompass one or several child sessions, is moved across the two domains. It can be seen from the figure that the actor has two relationships, or role-sessions, RS1 and RS2 with Server1 and GenericRole, respectively. The GenericRole actor is offering services which are available only within domain1, while Server1 provides global services.

When an actor is moved, as pointed out in section 3.2.2, all the parts that constitutes the actor must be recreated or recovered at the new location. However, certain elements might not be recoverable or have lost its relevance. For instance, certain capabilities might not be available at the new location, or a role-session is no longer relevant (as is the case for the relationship with the domain-specific GenericRole in the two domains).

The MobilityManager (MM) within a domain (or sub-domain) is always responsible for managing the accessibility to this actor, thus, the MM must be notified of the actors new location once the actor is up-and-running at that location. The actor will notify the MM with a LocationUpdate request. Requests to actors are always preceded with an ActorDiscovery requests sent to the local MM, thus providing up-to date location information to all movable actors.

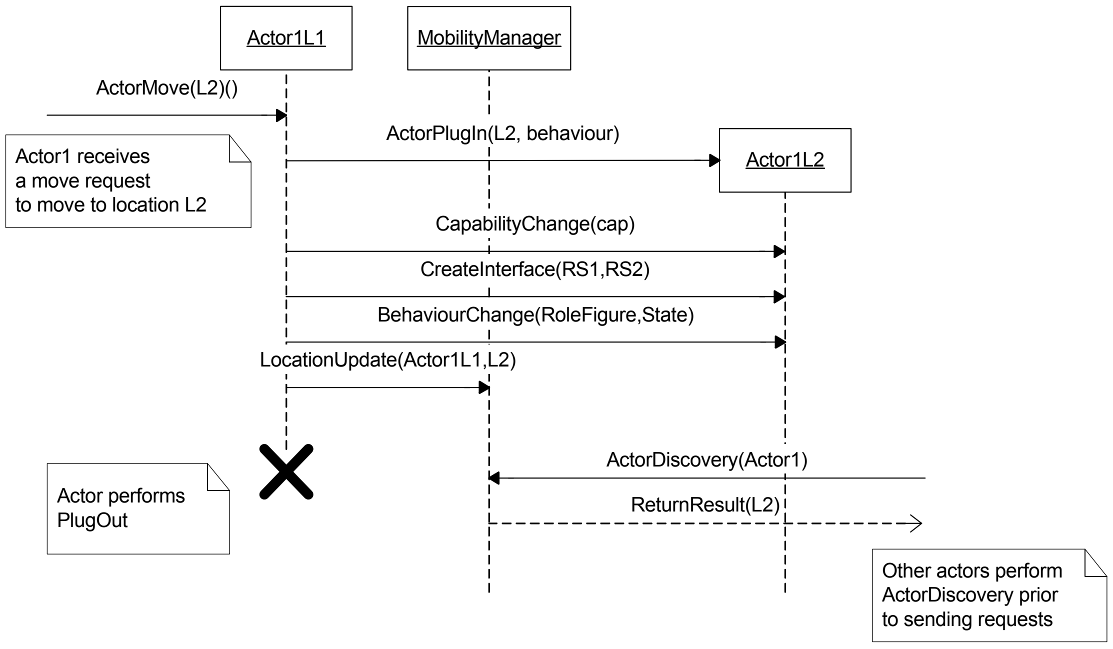

Actor1L1
MobilityManager
ActorMove(L2)()
ActorPlugIn(L2, behaviour)
Actor1L2
Actor1 receives a move request to move to location L2
CapabilityChange(cap)
CreateInterface(RS1,RS2)
BehaviourChange(RoleFigure,State)
LocationUpdate(Actor1L1,L2)
ActorDiscovery(Actor1)
Actor performs PlugOut
ReturnResult(L2)
Other actors perform ActorDiscovery prior to sending requests

Figure 3-10: Message sequence of a general event-driven ActorMove procedure

Figure 3-10 [MALM4, page 13] illustrates a possible message sequence of an actor changing and updating its location, while some node performs an ActorDiscovery procedure looking up for this actor. Upon receiving the move request, ActorMove, a series of actions need to be performed to re-instantiate the actor instance at the new location. Firstly, an ActorPlugIn procedure is carried out, then recovering of capabilities, role-sessions, behaviour should follow via CapabilityChange, CreateInterface, and BehaviourChange procedures, respectively. Secondly, the MobilityManager is updated via a LocationUpdate procedure. This particular example is based on an event driven ActorMove (i.e. the actor is explicitly commandeered to move), however, performing certain types of checks at specified intervals (i.e. check that required capabilities does not degrade below an acceptable level), could trigger an implicit ActorMove procedure.

3.4.3 Actor mobility scenarios

When moving an actor, there are four possible scenarios, and, as was the case for terminal mobility, these scenarios have different levels of complexity. The different scenarios are based on at what level in the hierarchy the move is affecting the controlling entities. For example, a move within the node does not affect the local MobilityManager in any degree, while a move between two domains affect not only the two MMs but also the Directors in charge of each domain.

3.4.3.1 Within same node/terminal

Although it does not make much sense to have an actor move within the same terminal, this scenario has been included to further emphasise the flexibility of the actor mobility procedures, as well as providing for easier testing and verification of the model.

3.4.3.2 Within same sub-domain

This is the most effortless of the inter-terminal actor move scenarios, and occurs when an actor is moved between two terminals being governed by the same MobilityManager (and Director).

3.4.3.3 Between two sub-domains in the same domain

When an actor is moved between two sub-domains in the same domain, the governing MM of the first domain is exchanged with the MM of the second domain, but the Director entity will be the same.

3.4.3.4 Between two domains

The most complex form of an actor move procedure occurs when an actor is moved between two domains, and both the MM and Director is exchanged with ‘new’ ones.

4 Implementation of mobility

The MicroTAPAS architecture was realised and implementation using the Java2 Micro Edition (J2ME) programming language from Sun Microsystems, and is described in [LUHE].

This chapter will present an overview of the actual implementation of the mobility support functionality for TAPAS which has been discussed in the immediate previous chapters. The chapter will guide the reader through all major parts of the implementation and try to convey to the reader the accomplishments made.

During the development of the mobility extension, it became apparent that the structure of the implementation would benefit from being split into relevant java packages, instead of piling every class and interface into one package. Thus, three sub-packages were included, each corresponding to a well defined area of operation;

- `MicroTAPAS.debug` contains classes and interfaces used when debugging the system, such as a dedicated debug server. The debug package is described in section 4.6.1.

- `MicroTAPAS.mobility` is the main package, and contains all the relevant classes for the mobility extension of MicroTAPAS.

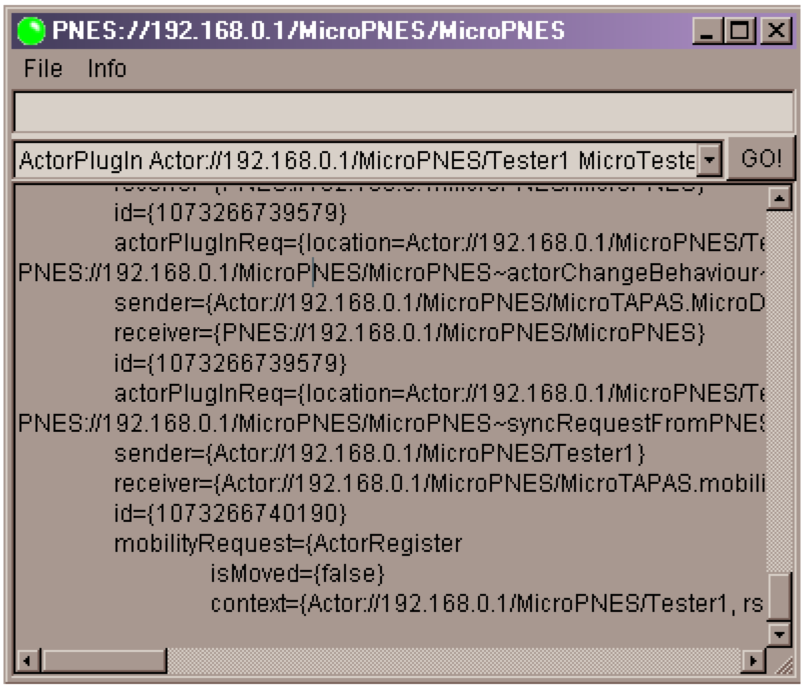
- `MicroTAPAS.util` contains a small collection of utility classes which are available to all parts of the system.

The main focus of this chapter is the mobility package.

4.1 Overview

At first glance, the MicroTAPAS support architecture is very similar to the standard TAPAS software package, and consists of two integral parts. The first part is a small-footprint component located at each node whishing to participate in a TAPAS network, and is referred to as the bootstrap. A typical size (depending on the number attributes in the configuration files) for the bootstrap lies around 10 KB. The second part of the package is a software library located at a centrally known location (conveyed to the bootstrap through parameters in the configuration file). When a TAPAS node is 'fired up' the bootstrap connects, through the network interface, to the centrally located repository, and dynamically downloads all required TAPAS software components.

Most of the user communication with the support architecture, at least in the early stages after a fresh start-up, happens through the MicroTAPAS debug GUI, depicted in Figure 4-1. The user can execute TAPAS specific commands from either the text-field or from the pull-down menu. This is an enhancement tailored specifically for PDA's and their limited input capability. Result from the system is displayed in the large text-area in the centre of the GUI.


PNES://192.168.0.1/MicroPNES/MicroPNES
File
Info
ActorPlugIn Actor://192.168.0.1/MicroPNES/Tester1 MicroTeste
GO!

```
        id={1073266739579}
        actorPlugInReq={location=Actor://192.168.0.1/MicroPNES/Te
PNES://192.168.0.1/MicroPNES/MicroPNES~actorChangeBehaviour
        sender={Actor://192.168.0.1/MicroPNES/MicroTAPAS.MicroD
        receiver={PNES://192.168.0.1/MicroPNES/MicroPNES}
        id={1073266739579}
        actorPlugInReq={location=Actor://192.168.0.1/MicroPNES/Te
PNES://192.168.0.1/MicroPNES/MicroPNES~syncRequestFromPNE
        sender={Actor://192.168.0.1/MicroPNES/Tester1}
        receiver={Actor://192.168.0.1/MicroPNES/MicroTAPAS.mobili
        id={1073266740190}
        mobilityRequest={ActorRegister
                isMoved={false}
                context={Actor://192.168.0.1/MicroPNES/Tester1, rs
```

Figure 4-1: Main MicroTAPAS user interface/debug GUI

Most TAPAS entities will have an instance of the main debug window, and is very convenient for direct communication with the various TAPAS elements.

4.2 Mobility entities

The various entities which were engineered for the mobility extension to TAPAS are quite complex and have thus been divided into sub-sections, and are dealt with in an orderly fashion.

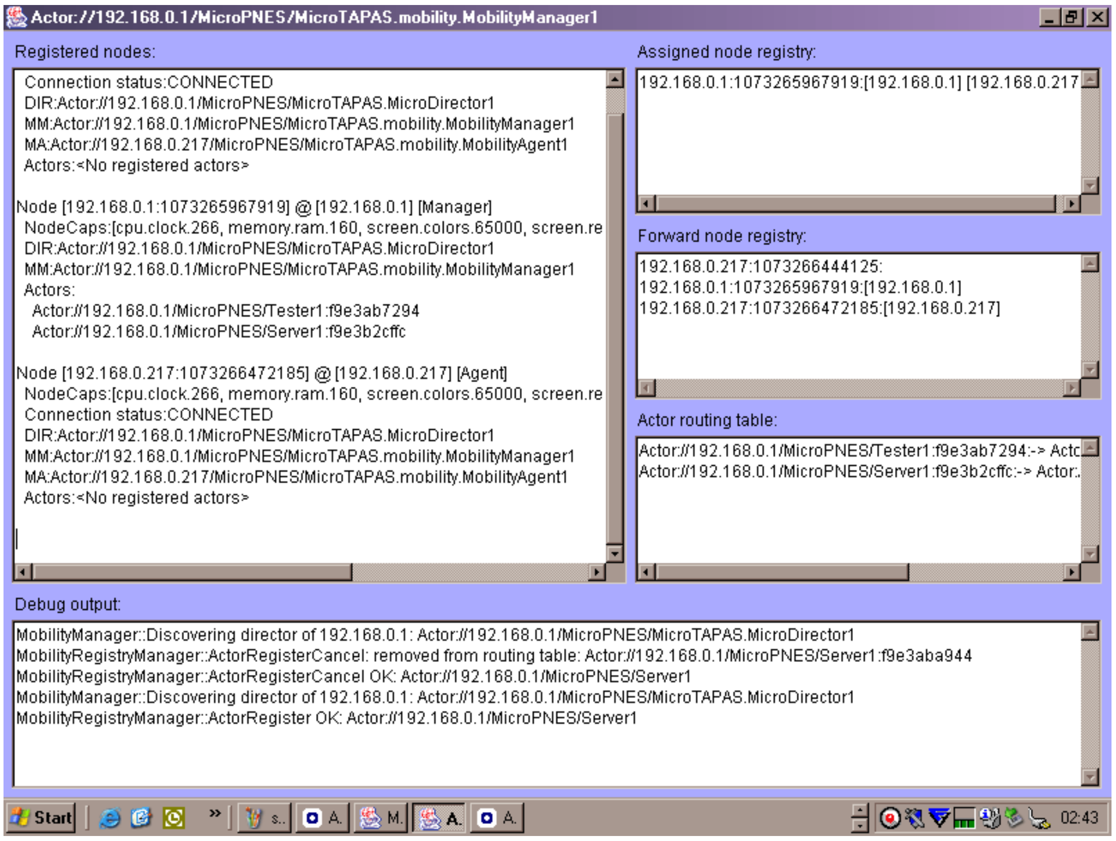
4.2.1 MobilityManager and MobilityAgent

The MobilityManager (MM) and MobilityAgent (MA) are the two most central parts of the mobility extension to MicroTAPAS. These two classes contain some of the same functionality, which was put in a class MobilitySupportActor (MSA), and MM and MA inherits this class. MSA contains the MicroPingClient and MicroPingServer classes, presented in section 4.2.4.

The MM and MA is responsible for handling all mobility related administration, such as;

- Keeping track of actors and terminals, including notifying/updating each other about registered actors and terminals.
- Facilitate in ActorMove and TerminalMove operations.
- Monitoring connection status to the MM (for MA nodes) and to all registered actor nodes (for MM nodes).

The MM also contains a MobilityManagerFrame which provides the user with an overview of all connected actors and terminals.



```
Connection status:CONNECTED
 DIR:Actor://192.168.0.1/MicroPNES/MicroTAPAS.MicroDirector1
 MM:Actor://192.168.0.1/MicroPNES/MicroTAPAS.mobility.MobilityManager1
 MA:Actor://192.168.0.217/MicroPNES/MicroTAPAS.mobility.MobilityAgent1
 Actors:<No registered actors>

Node [192.168.0.1:1073265967919] @ [192.168.0.1] [Manager]
 NodeCaps:[cpu.clock.266, memory.ram.160, screen.colors.65000, screen.re
 DIR:Actor://192.168.0.1/MicroPNES/MicroTAPAS.MicroDirector1
 MM:Actor://192.168.0.1/MicroPNES/MicroTAPAS.mobility.MobilityManager1
 Actors:
  Actor://192.168.0.1/MicroPNES/Tester1:f9e3ab7294
  Actor://192.168.0.1/MicroPNES/Server1:f9e3b2cffc

Node [192.168.0.217:1073266472185] @ [192.168.0.217] [Agent]
 NodeCaps:[cpu.clock.266, memory.ram.160, screen.colors.65000, screen.re
 Connection status:CONNECTED
 DIR:Actor://192.168.0.1/MicroPNES/MicroTAPAS.MicroDirector1
 MM:Actor://192.168.0.1/MicroPNES/MicroTAPAS.mobility.MobilityManager1
 MA:Actor://192.168.0.217/MicroPNES/MicroTAPAS.mobility.MobilityAgent1
 Actors:<No registered actors>
```

```
192.168.0.1:1073265967919:[192.168.0.1] [192.168.0.217
```

```
192.168.0.217:1073266444125:
192.168.0.1:1073265967919:[192.168.0.1]
192.168.0.217:1073266472185:[192.168.0.217]
```

```
Actor://192.168.0.1/MicroPNES/Tester1:f9e3ab7294:-> Actc
Actor://192.168.0.1/MicroPNES/Server1:f9e3b2cffc:-> Actor:
```

```
MobilityManager::Discovering director of 192.168.0.1: Actor://192.168.0.1/MicroPNES/MicroTAPAS.MicroDirector1
MobilityRegistryManager::ActorRegisterCancel: removed from routing table: Actor://192.168.0.1/MicroPNES/Server1:f9e3aba944
MobilityRegistryManager::ActorRegisterCancel OK: Actor://192.168.0.1/MicroPNES/Server1
MobilityManager::Discovering director of 192.168.0.1: Actor://192.168.0.1/MicroPNES/MicroTAPAS.MicroDirector1
MobilityRegistryManager::ActorRegister OK: Actor://192.168.0.1/MicroPNES/Server1
```

Figure 4-2: MobilityManagerFrame screenshot

4.2.2 MobilityApplicationActor

The MobilityApplicationActor is the actor side of the mobility extension, and provides the actors with mobility support functions. This class is an extension of the MicroApplicationActor.

The five most important functions in this class are the following:

- *actorRegister* and *actorRegisterCancel* register and cancel a registration of the actor with the nodes MobilityManager.
- *actorMove* method to move this actor to another location.
- *actorChangeBehaviour* changes the behaviour of this instance to that of another.
- *sendCreateInterface* is used together with actorMove to re-create the actors interface at the new location.

4.2.3 Mobility requests

A number of new MobilityRequest types were introduced with the mobility extension of MicroTAPAS, and is used for handling the various aspects of mobility.

MobilityRequest type	**Description**
ActorRegister	Registers an actor with the MobilityManager.
ActorRegisterCancel	Cancels the registration of an actor with the MobilityManager.
ActorRegisterMove	Cancels the actor registration with a MobilityManager when an actor has been moved to a different node.

NodeRegister	Registers a node with the MobilityManager. This is done automatically at start-up.
NodeRegisterCancel	Cancels a registration with the MobilityManager when a node is either moved to another MobilityManager, or if the node is about to shut down.
ActorDiscovery	Discovers an actor.
NodeDiscovery	Discovers a node.
DirectorDiscovery	Discovers a director.
MMDiscovery	Discovers a MobilityManager.
NodeValidate	Check if the node is registered with the current MobilityManager.
NodeRegisterRequest	A request sent to a node which is found not to be registered. This node will then react by sending a NodeRegister request.
TerminalMove	Used when a terminal is moved.
MobilitySync	Used when MobilityManagers send synchronisation messages between each other to update the tables of registered actors and nodes.
CapabilityValidate	Validates the requested capabilities of an actor against the offered capabilities of a node.
NodeCapabilityChange	Used when a node's capabilities are changed.
ActorMove	Used during an actor move procedure.
ActorCreateInterface	(Re-)creates an actor's interface at a new location.
RoleSessionUpdate	Updates the role-sessions at a director after an actor, or terminal, has moved to a new location.
RTT	Used to measure the RTT (Round-Trip-Time) between two nodes in a MicroTAPAS network.

Table 1: MobilityRequest types

4.2.4 Ping client and ping server

These entities were initially introduced in [LUHE, page 16] and are used in two different modes; a) to verify a node's network connection, and b) to verify that a registered actor is accessible.

The PingServer is a passive entity which simply waits for a connection request on a predefined port (configurable in the node configuration file), once a request is made, the server accepts the request and immediately closes the connection down. This gives the PingClient a confirmation that the 'pinged' node is alive and connected to the network.

The PingClient operates in one of two modes, depending on whether it is a MM or MA node. If the node is a MM node, it will obtain a list of all actors registered at the node and send a ping request to each of them. If a node is found to be unavailable (the node might be out-of-coverage, or has shut down), its status is set to 'not connected', and any attempt to route messages to that node will fail. The interval between pings is static, and is configurable in the node configuration file.

A MA node, on the other hand, will only ping its own MM node, and is used to verify its own connection status to the network. If the node is found to be disconnected, and depending on the type of actors running at the node, and their configuration, a number of different actions can be taken. The interval between pings is determined dynamically, where loss of connection will cut the current interval in half, and a online connection of more than ten times the initial interval will increase the current interval by the initial interval.

4.2.5 Capability monitor agent

The capability monitor agent, or CapsMonitorAgent, is active in all MobilityApplicationActors, and is responsible for monitoring the offered capabilities of a node, and compare them with the actor's required capabilities. If the offered capabilities are found to be 'less' than the required, an actorMove procedure will be set in motion. The interval between the checks is configurable in the actor configuration file.

4.3 Class diagrams

The simplified class diagram of the mobility package is shown in Figure 4-3. The sheer size of the classes with their respective variables and methods was too large to include here in the main text. Please refer to Appendix C for a complete listing of classes, methods and variables.

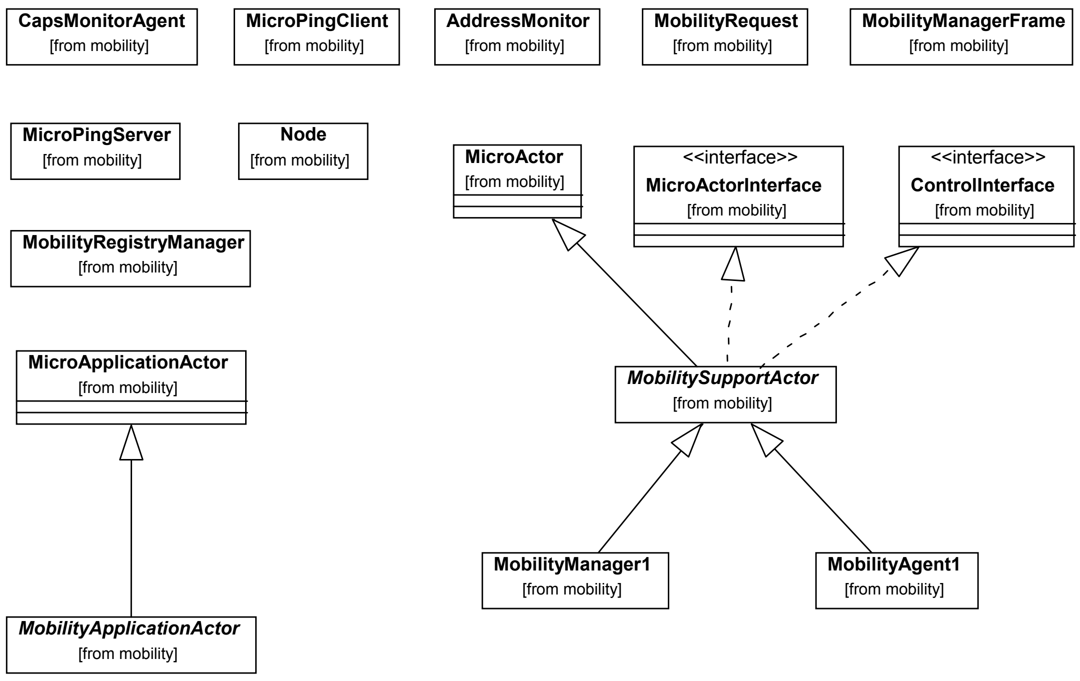

CapsMonitorAgent [from mobility]
MicroPingClient [from mobility]
AddressMonitor [from mobility]
MobilityRequest [from mobility]
MobilityManagerFrame [from mobility]
MicroPingServer [from mobility]
Node [from mobility]
MicroActor [from mobility]
<<interface>> MicroActorInterface [from mobility]
<<interface>> ControlInterface [from mobility]
MobilityRegistryManager [from mobility]
MicroApplicationActor [from mobility]
MobilitySupportActor [from mobility]
MobilityManager1 [from mobility]
MobilityAgent1 [from mobility]
MobilityApplicationActor [from mobility]

Figure 4-3: Simplified class diagram of the mobility package for MicroTAPAS

4.4 Capabilities

A simple system of capabilities were introduced and implemented in this prototype so as to provide the support architecture with a more realistic environment in which to operate. Capabilities are referred to in two different settings; *required* capabilities and *offered* capabilities. The required capabilities are those capabilities *required* by an actor and are matched with those capabilities *offered* by a node.

Prior to processing any ActorPlugIn request, the node on which the actor is expected to be plugged in verifies that the nodes offered capabilities are equal to, or better, than those capabilities required by the actor. If the node fails to meet the requirement, the plug in of the actor will ultimately fail as well, and the initiating entity will receive a notification of this. The ActorPlugIn request is an integral part of the ActorMove procedure, and, thus, capabilities are matched during the execution of that procedure as well.

Along with being checked prior to plugging in, the offered capabilities of a node are also monitored by the various plugged in actors during their lifetime. If a nodes capability is found to have deteriorated below the level expected by the actor, the actor will initialise an ActorMove procedure to a location specified by that actors configuration file.

Capability specifications may be done as a dotted (i.e. '.') separated list of terms, where each term represents a level in the hierarchical specification of the capability, i.e. 'printer' defines the capability 'any printer', while 'printer.postscript' specifies the capability 'printer with postscript capabilities'. The term asterisk (i.e. '*') may be used to specify 'any value' for a capability term, i.e. 'printer.*' specifies any printer capability.

In this prototype version, six capabilities were introduced; 'architecture', 'cpu.clock', 'memory.ram', 'screen.colors', 'screen.resolution.x', and 'screen.resolution.y'. Each of these capabilities can have either a numerical value, i.e. capability 'memory.ram.64' signifies 64 mega bytes (MB) of internal memory, or an asterisk, to denote that any value is acceptable. All the required capabilities of an actor are stored in a file, while a node's capabilities are either stored in a file, or determined dynamically. In this version, only 'resolution.x' and 'resolution.y' are determined dynamically, while the others are located in the text file. Refer to section 4.5 for an overview of the different types of data files and their location.

4.5 Data files

A number of new data files have been introduced to ease the configuration of the MicroTAPAS support system. The files are split into two categories; one set of files which configure the node and another set of files for configuring each actor individual.

The capability model and example used here is regarded as a very simple and straightforward one, a more elaborated platform is being developed and has yet to be tested, and that is why it has not been tested as part of the MicroTAPAS support platform. As any future continuation of this task it is suggested to shift to an overall capability support system that is capable of handling more complex issues regarded capability registry, update, advertisement, management, etc.

4.5.1 Node data files

Each node in a MicroTAPAS network is required to contain two data files in its bootstrap root directory; a configuration file and a capability file, tapas.cfg and node.cap, respectively. In addition to these two, a third, optional, scripting file can be added; node.script.

4.5.1.1 Configuration file

The idea of using a configuration file in TAPAS is not new, and is standard in the basic support architecture. With the introduction of MicroTAPAS, the file was heavily modified and more attributes were inserted, along with the possibility to included comments, among other things, and is described in [LUHE, page 18]. There are several new attributes in the configuration file this time around, most of them concerned with mobility. See Appendix B.1.1 for an example configuration file.

4.5.1.2 Capability file

Although part of the original TAPAS architecture specification, a working implementation of capabilities has never been released. A node's *offered* capabilities are included in the file `node.caps`, located in the bootstrap root directory. Table 2 shows an example capability file. This particular example is taken from a PDA with an ARM CPU running at 240 MHz and 64 MB of internal memory and a screen resolution of 240x320 with 65 thousand colours.

```
architecture.arm
cpu.speed.210
memory.64
screen.resolution.x.240
screen.resolution.y.320
screen.color.65000
```

Table 2: Example of a node's offered capabilities: node.caps

4.5.1.3 Script file

The concept of a script file is new to TAPAS, and this feature was initially included to help in testing the system. Each line in the text file corresponds to a TAPAS request, and, depending on the 'runscriptauto' attribute in the configuration file, is automatically executed sequentially each time the support architecture has been started on a node. Alternatively, the script can be started manually by selecting 'Start script' from the File menu of the TNES GUI. Table 3 contains two lines from an example script file. Each line is a separate command consisting of four attributes, where each attribute is delimited with a '¤' character. The first attribute is the type of TAPAS request to be sent, in this case PlayPlugIn and ActorPlugIn. The second attribute is the attributes of the TAPAS request, where <codebase> and <selfNode> automatically is replaced by is real values. The third attribute is used for logging purposes and says something about whether the operation is expected to be a success or not. This is significant if one has a long script executing and its difficult to quickly scan the log for any potential problems. If you perform an operation that is not expected to be successful, for example try to plug in an actor before having plugged in its corresponding play, one would set this attribute to false, then the result of the operation would also be false, and the operation is a success, i.e. both the expected result and the real result must be either true or false for the operation to be marked as a success in the log. The last attribute is the time, in milliseconds, for which the script handler should wait before executing this particular command.

```
PlayPlugIn ¤ MicroTester v2_0 <codebase> ¤ true ¤ 0

ActorPlugIn ¤ Actor://<selfNode>/MicroPNES/Tester1 MicroTester1 ¤ true ¤ 0
```

Table 3: Example node script

Throughout development and testing of the Mobility extensions for MicroTAPAS, the use of scripting has proved a very helpful tool. However, its an good way to consistently perform repetitive tasks, either at startup time, or during normal operation of a node.

4.5.2 Actor data files

Actors now have built in support for being configured at runtime by using a configuration file ('.cfg' extension) and a capability file ('.cap' extension). These files can be located either at the codebase (central location for all TAPAS runtime components) or at each individual node. If the files are located at both locations, the local files have precedence over foreign files.

The files are normal UTF-8 encoded text files, and are named by their actor role, followed by the configuration or capability extension:
<actorRole>.cap
<actorRole>.cfg

The foreign files are located at the codebase, and conform to the naming scheme mentioned above:
<codebase>/<actorRole>.<[cap|cfg]>

Local files are located in a hierarchy below the tapasroot and dataroot directories at each node (which in turn are configurable in the node configuration file):
<tapasroot><dataroot><playName>/<playVersion>/<actorRole>.<[cap|cfg]>

This naming and location scheme conforms to the standard TAPAS scheme and provides a sound basis for naming and locating these types of files.

4.5.2.1 Configuration file

With this extension of the basic TAPAS architecture, it was decided that the ability to configure actors at runtime would be very beneficial for the system as a whole, instead of recompiling actors each time a parameter was changed or manually insert the parameters each time an actor was plugged in.

So far, only two attributes have been included in the actor configuration file; movestrategy and movelocation. Both of these attributes deal with mobility, where the first attribute, movestrategy, decides which strategy to use when an actor is moved, see section 3.4.2. Table 4 gives an example of an actor configuration file.

```
# Variable that decides what strategy to use when actor should be moved
# alternatives are: 'a', 'b' or 'c' ('a' is the most simple)
movestrategy = a

# Default location to move actor to, if offered capabilities deteriorate
movelocation = PNES://10.0.0.1/MicroPNES/MicroPNES
```

Table 4: Example actor configuration file: <actorRole>.cfg

Actor configuration files can be located in two different locations; at the central codebase location, where actors and manuscripts are located, and/or at each individual node. If the configuration files are located at both locations, the local files take precedence over the centrally stored files.

4.5.2.2 Capability file

Whereas a node must offer some sort of capability, it is optional for an actor to require an number of capabilities. An actor's required capability/capabilities are located in a text file, and, as for the actor configuration files, can be located either at a centrally known location or at each individual node. Locally stored capabilities, if present, takes precedence over the required capabilities stored at the foreign location.

Typically, if an actor does have any requirements, these will indicate the least available resources under which it can successfully execute. For example, a debugging actor tool which includes a large debugging info window, displaying lots of information, would probably be unsuitable for devices with small screens. Likewise, an actor tasked with lots of complex mathematical computation would be unsuited for a device with limited computational power.

An actor's required capabilities are inserted into the ActorPlugInRequest when handled by the director, before being forwarded to the node where the actor is supposed to be plugged in. When the request is received by the intended recipient, the actor's required capabilities are matched with the nodes offered capabilities, and, if successful, the plug in can proceed, if not, the plug in has failed, and the initiating party will receive a notification about the failure and its cause.

4.6 Various enhancements and additions

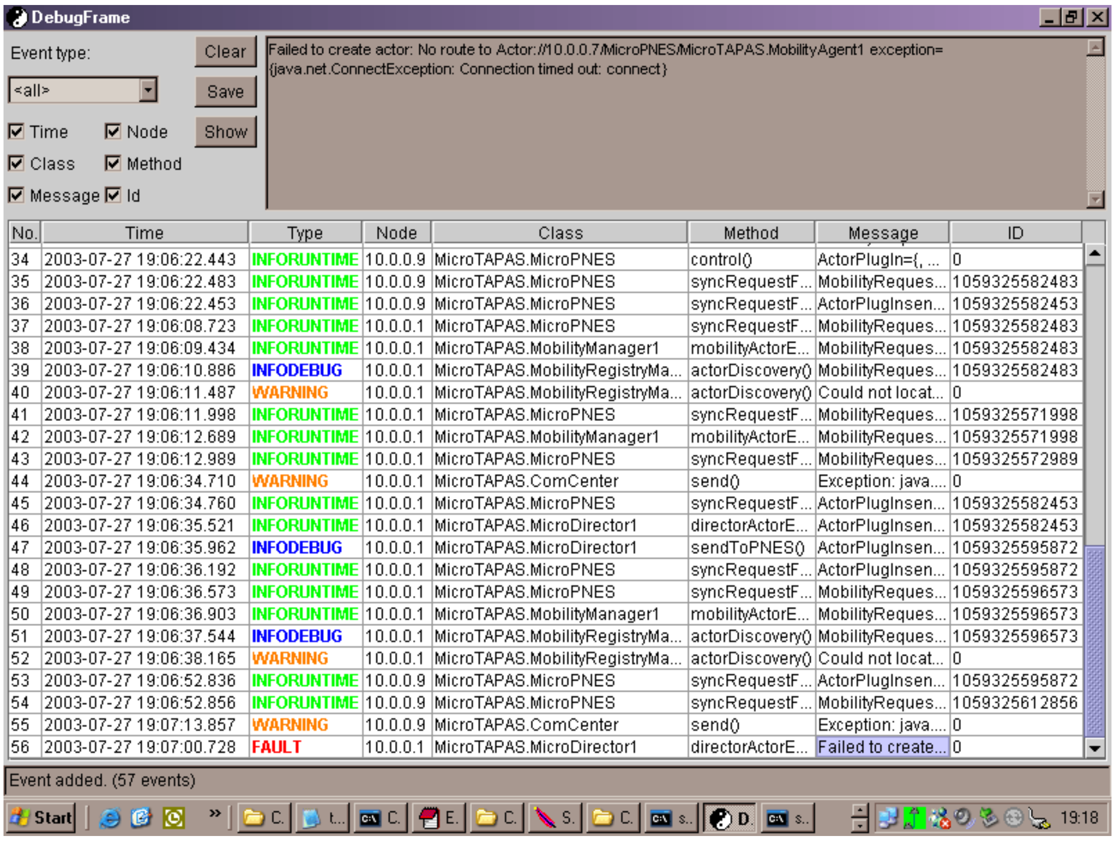
In addition to the implemented extensions mentioned above, a few general enhancements have also been added to the MicroTAPAS support architecture.

4.6.1 Debug package

A debug package was developed to help performing various debugging tasks in the MicroTAPAS support architecture. The main feature of this package is the debug server, and the reporting of various debugging events to this server. The server can be started on any TAPAS enabled terminal, and it will accept debugging events from all the TAPAS entities in a network. A DebugEvent can be of one of the following event types; FATAL, FAULT, WARNING, INFORUNTIME, INFODEBUG. In addition to the event type, each event contains information about the node, class, method, timestamp, and an event message.

Every MicroTAPAS actor has an instance of a small debugging object, Debug. It is a simple class to only facilitate sending DebugEvents to the DebugServer. The actual sending of the events are done using a thread, because otherwise the system would hang for as long as the object is trying to connect to the server. Once the connection to the server fails, all subsequent calls to this instance will fail, indicating to the system that the server is unavailable. See Appendix C for Javadoc on the debug package.

Figure 4-4 shows an example screenshot from the DebugServer GUI. The GUI is divided into three main sections; options, text-view and the list of events. It can be seen from the picture that there are a number of different options to choose from. The "Event type" drop-down box lets users decide which event types should be displayed, ie. one can choose to display only WARNING or FAULT event types. The collection of six checkboxes determines what event information is displayed, and any one of these can be turned on or off, according to the whish of the user. The tree buttons, Clear, Save and Show are currently not implemented – but could be programmed to clear, save or show (load) the list of events. The large text area is for displaying text that can not fit in a cell in the list of events (the currently shown text is whichever cell is highlighted). The largest portion of the screen is occupied by the list of events, and they are listed in the order they are received by the debug server.



Failed to create actor: No route to Actor://10.0.0.7/MicroPNES/MicroTAPAS.MobilityAgent1 exception={java.net.ConnectException: Connection timed out: connect}

No.	Time	Type	Node	Class	Method	Message	ID
34	2003-07-27 19:06:22.443	INFORUNTIME	10.0.0.9	MicroTAPAS.MicroPNES	control()	ActorPlugIn={, ...	0
35	2003-07-27 19:06:22.483	INFORUNTIME	10.0.0.9	MicroTAPAS.MicroPNES	syncRequestF...	MobilityReques...	1059325582483
36	2003-07-27 19:06:22.453	INFORUNTIME	10.0.0.9	MicroTAPAS.MicroPNES	syncRequestF...	ActorPlugInsen...	1059325582453
37	2003-07-27 19:06:08.723	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroPNES	syncRequestF...	MobilityReques...	1059325582483
38	2003-07-27 19:06:09.434	INFORUNTIME	10.0.0.1	MicroTAPAS.MobilityManager1	mobilityActorE...	MobilityReques...	1059325582483
39	2003-07-27 19:06:10.886	INFODEBUG	10.0.0.1	MicroTAPAS.MobilityRegistryMa...	actorDiscovery()	MobilityReques...	1059325582483
40	2003-07-27 19:06:11.487	WARNING	10.0.0.1	MicroTAPAS.MobilityRegistryMa...	actorDiscovery()	Could not locat...	0
41	2003-07-27 19:06:11.998	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroPNES	syncRequestF...	MobilityReques...	1059325571998
42	2003-07-27 19:06:12.689	INFORUNTIME	10.0.0.1	MicroTAPAS.MobilityManager1	mobilityActorE...	MobilityReques...	1059325571998
43	2003-07-27 19:06:12.989	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroPNES	syncRequestF...	MobilityReques...	1059325572989
44	2003-07-27 19:06:34.710	WARNING	10.0.0.1	MicroTAPAS.ComCenter	send()	Exception: java....	0
45	2003-07-27 19:06:34.760	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroPNES	syncRequestF...	ActorPlugInsen...	1059325582453
46	2003-07-27 19:06:35.521	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroDirector1	directorActorE...	ActorPlugInsen...	1059325582453
47	2003-07-27 19:06:35.962	INFODEBUG	10.0.0.1	MicroTAPAS.MicroDirector1	sendToPNES()	ActorPlugInsen...	1059325595872
48	2003-07-27 19:06:36.192	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroPNES	syncRequestF...	ActorPlugInsen...	1059325595872
49	2003-07-27 19:06:36.573	INFORUNTIME	10.0.0.1	MicroTAPAS.MicroPNES	syncRequestF...	MobilityReques...	1059325596573
50	2003-07-27 19:06:36.903	INFORUNTIME	10.0.0.1	MicroTAPAS.MobilityManager1	mobilityActorE...	MobilityReques...	1059325596573
51	2003-07-27 19:06:37.544	INFODEBUG	10.0.0.1	MicroTAPAS.MobilityRegistryMa...	actorDiscovery()	MobilityReques...	1059325596573
52	2003-07-27 19:06:38.165	WARNING	10.0.0.1	MicroTAPAS.MobilityRegistryMa...	actorDiscovery()	Could not locat...	0
53	2003-07-27 19:06:52.836	INFORUNTIME	10.0.0.9	MicroTAPAS.MicroPNES	syncRequestF...	ActorPlugInsen...	1059325595872
54	2003-07-27 19:06:52.856	INFORUNTIME	10.0.0.9	MicroTAPAS.MicroPNES	syncRequestF...	MobilityReques...	1059325612856
55	2003-07-27 19:07:13.857	WARNING	10.0.0.9	MicroTAPAS.ComCenter	send()	Exception: java....	0
56	2003-07-27 19:07:00.728	FAULT	10.0.0.1	MicroTAPAS.MicroDirector1	directorActorE...	Failed to create...	0

Event added. (57 events)

Figure 4-4: DebugServer screenshot

4.6.2 Communication model

The communication model has been greatly improved over the model presented in [LUHE, page 13]. While the model still utilises sockets as a mean of establishing transmission channels between nodes, its handling is much improved and is less resource demanding than the initial proposal. In essence, the entity in charge of handling all inter-node communication, the ComCenter, consists of two buffers and an open port, on which it is listening for incoming requests. All PNES instances (one for each node) in a TAPAS system has an instance of this entity. There is one buffer each for incoming and outgoing requests. An incoming request is, as soon as possible, handed over to the parent PNES. Likewise, and outgoing request is sent as soon as the underlying network will permit, and granted that the receiving party is available. The port on which all ComCenters are listening *must* be the same for the whole support system, and is configurable through the configuration file on each node.

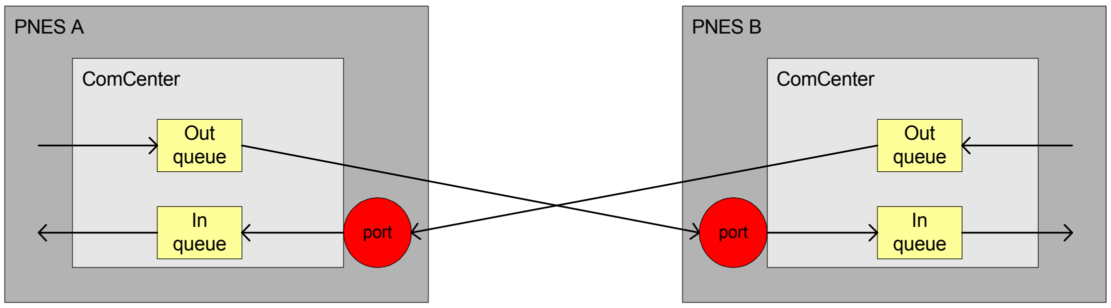


Figure 4-5: Communication model overview

Figure 4-5 illustrates the model with a simple diagram of two PNES instances, A and B, where both have an instance of the ComCenter entity.

5 Test application – MicroTester v.2

The tester application described in this chapter is version 2.0 of the test application described in [LUHE, page 23], and is an application specifically developed to allow testing of all implemented TAPAS support functionality. The new version of the tester included functionality that supports the new mobility extensions to MicroTAPAS, as well as a new feature to log executed commands and to display their result. The application consists of three application actors; MicroTester1, MicroTester2 and MicroTesterServer. The MicroTester1 and MicroTester2 actors are slightly modified versions of the same actor and are the user interface to the application and all commands will be executed thorough it's Graphical User Interface (GUI). The MicroTesterServer actor was included to have an actor that could be plugged in/out externally, as well as answer simple queries from the main actor. The following sections will in detail describe the various aspects of this application. Javadoc included in Appendix C.

5.1 Motivation

The development on the test application has been carried out in parallel to the development of the extended MicroTAPAS architecture. Each time a new support feature had been implemented in MicroTAPAS, a corresponding test-function was written in the test application to better aid in the further enhancement and debugging of the newly incorporated feature. Towards the end of the development cycle, the application was also extensively used to demonstrate and verify complex patterns of behaviour, often compromising two or more support functions at a time.

5.2 Functional requirements

- Upon start-up of the actor MicroTester (client) the user will be presented with a Graphical User Interface (GUI) for executing commands, and view status information, this is the main application window.
- All available commands should be accessible by menu items on a menu toolbar in the application window.
- The application will automatically plug out when the application window is closed.
- It should be possible to execute all of the standard TAPAS support functions; ActorPlugIn (server), ActorPlugOut (server), RoleSessionAction (server, client) and ActorChangeBehaviour (client).
- It should be possible to execute all the extended TAPAS support functions; ActorRegister (client), ActorRegisterCancel (client) and ActorMove (client).
- All performed actions should be logged.
- The logged actions can be printed to the application GUI.
- The log of performed actions can be cleared.
- When executing a command, the application window should display information to the user that indicates that work is in progress. This will ensure that time-consuming tasks will not be perceived as a lock-up.

5.3 Non-functional requirements

- The program shall be easy and intuitive to use. This includes extra thought given to the input of information on a device lacking a standard mouse and/or keyboard – i.e. a PDA having a touch-screen and a stylus.
- The program shall be compact and well written in order to minimize the necessary download time to, and memory footprint needed on, a handheld device.

5.4 Application overview

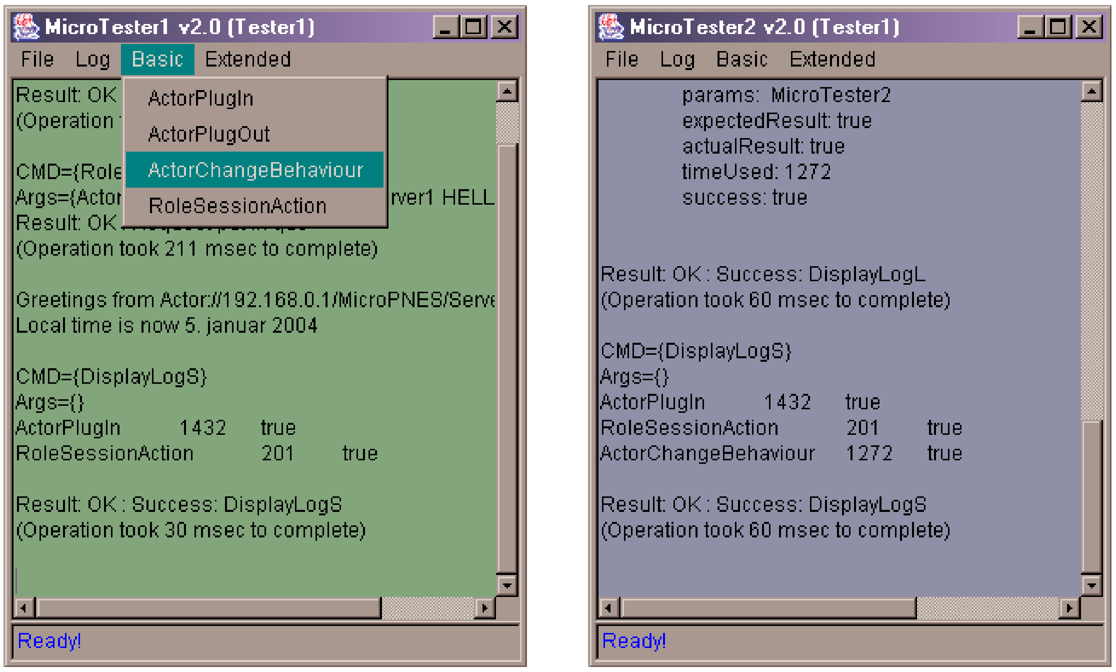
The MicroTester v.2 consists of three application actors; MicroTester1, MicroTester2 (clients) and MicroTesterServer (server). The client actors are slightly modified subclasses of MicroTesterBase, thus both inherit the main client GUI object; TesterMainWindow. The only difference between the two client actors is the background color of the main text area of the GUI (Tester1 is light green, while Tester2 is light blue), the reason for this will be discussed later in this section. The main GUI will be used to interact with the user of the application, and consists of three parts; the menu bar at the top, a text-output area in the middle, and a status line at the bottom of the window.

The menu bar contains four menus; File, Log, Basic and Extended. The File menu has two choices, Clear output, which clears the main text output area of the GUI, and Exit, which plugs out the actor and closes down the application. The Log menu has three choices; Show log (l), which displays a long version of the stored log, Show log (s) which displays a short version of the stored log, and Clear log, which clears the stored log from memory. The Basic menu contains the basic TAPAS support functions, and perform according to their names; ActorPlugIn, ActorPlugOut, ActorChangeBehaviour, RoleSessionAction. Likewise, the Extended menu contains the extended TAPAS support functions, and again behave according to their menu names; ActorRegister, ActorRegisterCancel and ActorMove.

The text output area of the GUI displays various bits of information to the user. This includes information about what action has been selected from the menus, what the result of the operation was and how long time it took to complete (shown in milliseconds). This text area is also used when the user wishes to see a list (long or short version) of the stored action log. The bottom status bar indicates the status of the application. When a command is issued from one of the menus, a 'working' indication is shown in this area.

5.5 Screenshots

Figure 5-1 shows a few different screenshots taken of the MicroTester v.2 application running at a laptop computer.



```
Result: OK
(Operation

CMD={Role
Args={Actor                                   rver1 HELL
Result: OK
(Operation took 211 msec to complete)

Greetings from Actor://192.168.0.1/MicroPNES/Serve
Local time is now 5. januar 2004

CMD={DisplayLogS}
Args={}
ActorPlugIn          1432      true
RoleSessionAction          201        true

Result: OK : Success: DisplayLogS
(Operation took 30 msec to complete)
```

```
          params:  MicroTester2
          expectedResult: true
          actualResult: true
          timeUsed: 1272
          success: true


Result: OK : Success: DisplayLogL
(Operation took 60 msec to complete)

CMD={DisplayLogS}
Args={}
ActorPlugIn          1432      true
RoleSessionAction          201        true
ActorChangeBehaviour      1272      true

Result: OK : Success: DisplayLogS
(Operation took 60 msec to complete)
```

Figure 5-1: Screenshots of MicroTester

5.6 Class diagram

This is a class diagram of the MicroTester v.2, and shows all the relevant classes and their relevant relationships.

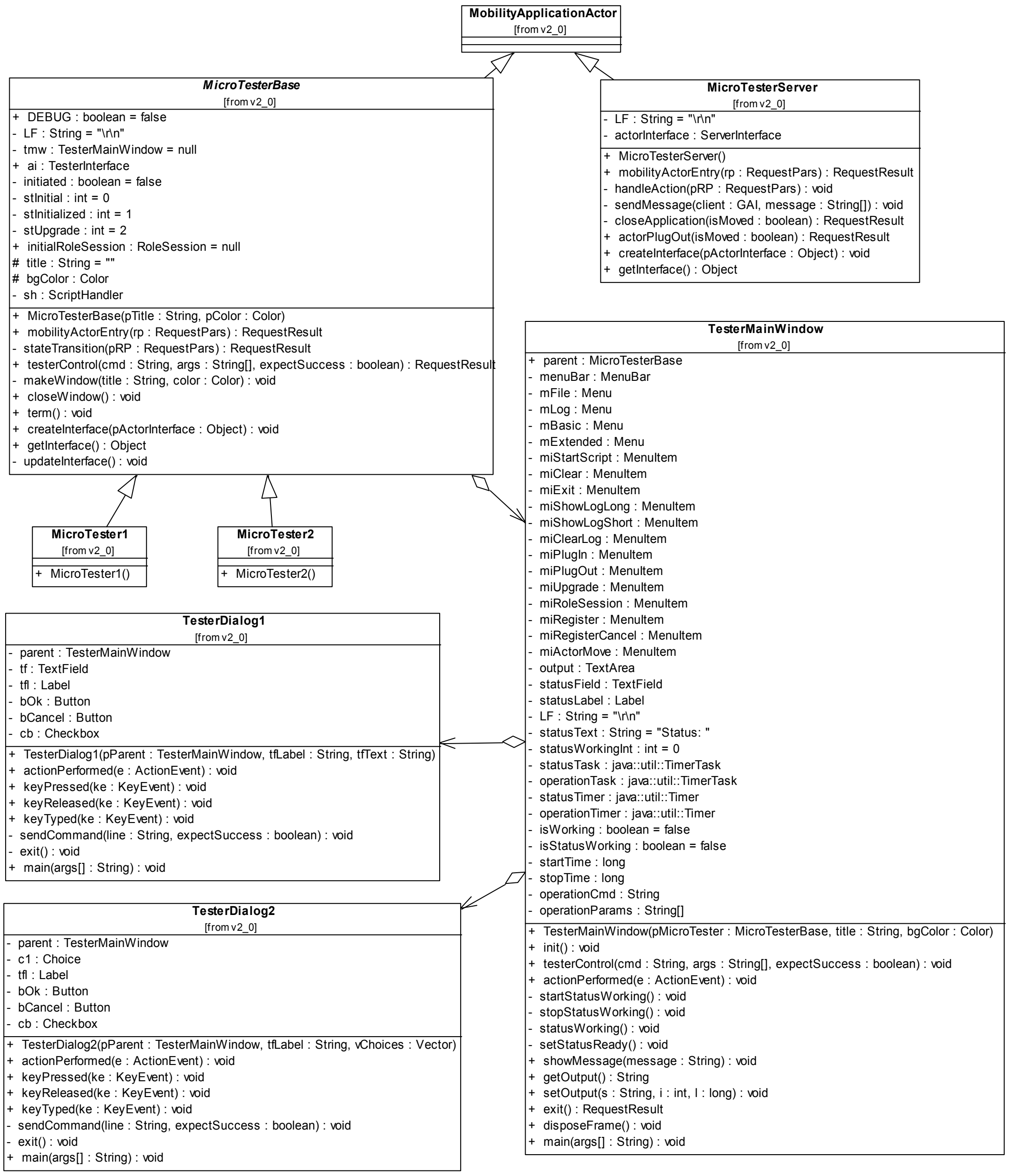

MobilityApplicationActor
[from v2_0]
MicroTesterBase
[from v2_0]
+ DEBUG : boolean = false
- LF : String = "\r\n"
- tmw : TesterMainWindow = null
+ ai : TesterInterface
- initiated : boolean = false
- stInitial : int = 0
- stInitialized : int = 1
- stUpgrade : int = 2
+ initialRoleSession : RoleSession = null
# title : String = ""
# bgColor : Color
- sh : ScriptHandler
+ MicroTesterBase(pTitle : String, pColor : Color)
+ mobilityActorEntry(rp : RequestPars) : RequestResult
- stateTransition(pRP : RequestPars) : RequestResult
+ testerControl(cmd : String, args : String[], expectSuccess : boolean) : RequestResult
- makeWindow(title : String, color : Color) : void
+ closeWindow() : void
+ term() : void
+ createInterface(pActorInterface : Object) : void
+ getInterface() : Object
- updateInterface() : void
MicroTesterServer
[from v2_0]
- LF : String = "\r\n"
- actorInterface : ServerInterface
+ MicroTesterServer()
+ mobilityActorEntry(rp : RequestPars) : RequestResult
- handleAction(pRP : RequestPars) : void
- sendMessage(client : GAI, message : String[]) : void
- closeApplication(isMoved : boolean) : RequestResult
+ actorPlugOut(isMoved : boolean) : RequestResult
+ createInterface(pActorInterface : Object) : void
+ getInterface() : Object
MicroTester1
[from v2_0]
+ MicroTester1()
MicroTester2
[from v2_0]
+ MicroTester2()
TesterMainWindow
[from v2_0]
+ parent : MicroTesterBase
- menuBar : MenuBar
- mFile : Menu
- mLog : Menu
- mBasic : Menu
- mExtended : Menu
- miStartScript : MenuItem
- miClear : MenuItem
- miExit : MenuItem
- miShowLogLong : MenuItem
- miShowLogShort : MenuItem
- miClearLog : MenuItem
- miPlugIn : MenuItem
- miPlugOut : MenuItem
- miUpgrade : MenuItem
- miRoleSession : MenuItem
- miRegister : MenuItem
- miRegisterCancel : MenuItem
- miActorMove : MenuItem
- output : TextArea
- statusField : TextField
- statusLabel : Label
- LF : String = "\r\n"
- statusText : String = "Status: "
- statusWorkingInt : int = 0
- statusTask : java::util::TimerTask
- operationTask : java::util::TimerTask
- statusTimer : java::util::Timer
- operationTimer : java::util::Timer
- isWorking : boolean = false
- isStatusWorking : boolean = false
- startTime : long
- stopTime : long
- operationCmd : String
- operationParams : String[]
+ TesterMainWindow(pMicroTester : MicroTesterBase, title : String, bgColor : Color)
+ init() : void
+ testerControl(cmd : String, args : String[], expectSuccess : boolean) : void
+ actionPerformed(e : ActionEvent) : void
- startStatusWorking() : void
- stopStatusWorking() : void
- statusWorking() : void
- setStatusReady() : void
+ showMessage(message : String) : void
+ getOutput() : String
+ setOutput(s : String, i : int, l : long) : void
+ exit() : RequestResult
+ disposeFrame() : void
+ main(args[] : String) : void
TesterDialog1
[from v2_0]
- parent : TesterMainWindow
- tf : TextField
- tfl : Label
- bOk : Button
- bCancel : Button
- cb : Checkbox
+ TesterDialog1(pParent : TesterMainWindow, tfLabel : String, tfText : String)
+ actionPerformed(e : ActionEvent) : void
+ keyPressed(ke : KeyEvent) : void
+ keyReleased(ke : KeyEvent) : void
+ keyTyped(ke : KeyEvent) : void
- sendCommand(line : String, expectSuccess : boolean) : void
- exit() : void
+ main(args[] : String) : void
TesterDialog2
[from v2_0]
- parent : TesterMainWindow
- c1 : Choice
- tfl : Label
- bOk : Button
- bCancel : Button
- cb : Checkbox
+ TesterDialog2(pParent : TesterMainWindow, tfLabel : String, vChoices : Vector)
+ actionPerformed(e : ActionEvent) : void
+ keyPressed(ke : KeyEvent) : void
+ keyReleased(ke : KeyEvent) : void
+ keyTyped(ke : KeyEvent) : void
- sendCommand(line : String, expectSuccess : boolean) : void
- exit() : void
+ main(args[] : String) : void

Figure 5-2: Class diagram of MicroTester v.2

5.7 Suggested improvements

Although the tester application can perform all of the implemented basic and extended support functions of MicroTAPAS, a few improvements could be included in later versions of the application. These suggested improvements are briefly described below.

- Implement a scheme to secure the communication between tester entities to prevent eavesdropping from an unauthorized third party.
- Add different levels of transparency, from a level similar to what is implemented in this version, to a very detailed view of all internal states, communication, requests and results.

- Extend the logging function to show statistics of all relevant pieces of information. This information could include number of connections, number of sent/received requests, detailed execution times etc.

6 Testing and performance

A full-scale and comprehensive test of the implemented mobility features of MicroTAPAS has, due to a few unresolved issues, not been possible. These issues will be described in a later section of this chapter. Each feature has, however, to a greater or lesser degree undergone testing to ensure that the extended support functionality was implemented according to the specifications put forward. A limited test was performed during a presentation at NTNU in September 2003, and this chapter describes the procedures.

6.1 Test Environment

The testing of the MicroTAPAS architecture, with the mobility extension was performed on a network set-up like described in Figure 6-1.

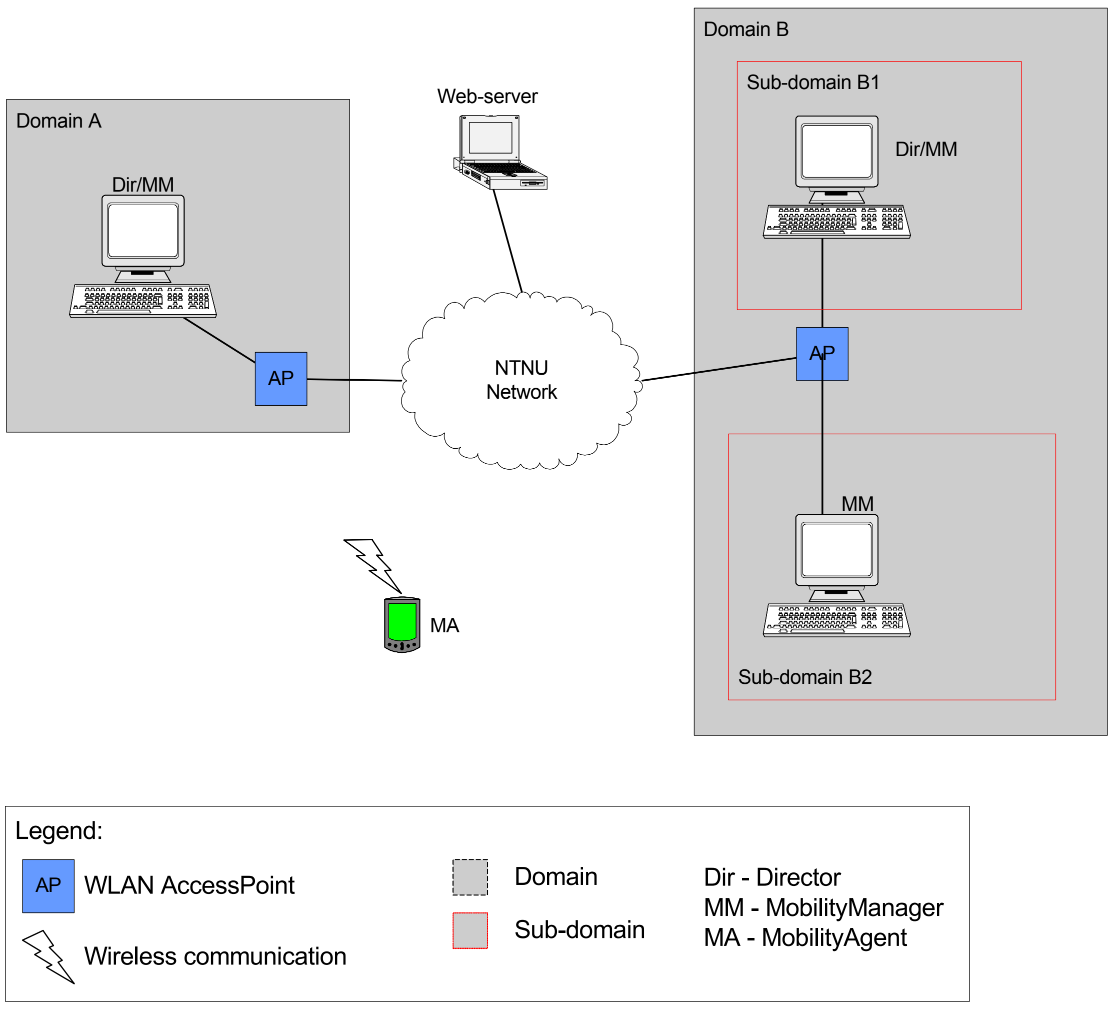


Figure 6-1: Test network

The network consisted of two wireless LAN access points, three stationary computers, one web server and one PDA. The used hardware is summarised in Table 5.

Node	Hardware	Connection	Virtual Machine (VM)	Role
A	Desktop PC	LAN	JVM	Dir/MM
B	Desktop PC	LAN	JVM	Dir/MM

C	Desktop PC	LAN	JVM	MM
D	PDA	WLAN	J9	MA
E	Laptop	LAN	--	Web-server

Table 5: Hardware components in test environment

6.2 Test procedures

To verify the mobility extension both the TerminalMove and ActorMove operations were performed. Three TerminalMove procedures and four ActorMove procedures were tested.

For the TerminalMove operation these procedures were tested:

a. Within same sub-domain (same MM)
b. Between two sub-domains (same Dir/different MM)
c. Between two domains (different Dir/MM)

According to the definition, TerminalMove is initiated when the terminal detects a change in the IP-address. There was, however, a problem when configuring the access points (AP), more specifically, the DHCP lease period of the addresses assigned to the terminal was set to 24 hours, thus moving from one ‘cell’ to another did not trigger a change in the address. The addresses thus had to be changed manually on the device, to simulate a move between two domains (coverage area of the APs). All role-sessions are updated to reflect the change of location.

For the ActorMove operation the following procedures were tested:

d. Within same terminal
e. Within same sub-domain (same Dir/MM)
f. Between two sub-domains (same Dir/different MM)
g. Between two domains (different Dir/MM)

ActorMove is initiated either by a degrading in a node’s offered capabilities, or by an explicit command.

6.3 Test results

During the demonstration, all seven test procedures (three TerminalMove and four ActorMove) were performed, and worked according to the definition.

6.4 Performance

No explicit performance test was carried out with this extended version of MicroTAPAS. A comprehensive performance test, however, was preformed in [LUHE, page 37]. The complete results from this test are included in Appendix B. The emphasis of the test was to clarify the longer execution times on a PDA compared to those on a regular PC (desktop and laptop) when running the TAPAS support architecture. The testing was carried out using a HeapTest program, available from Sun Microsystems Inc., and the PDA, unsurprisingly, was found to be several magnitudes slower than a conventional PC when it came to computational operations. The summary of the test is shown in Figure 6-2. Three terminals were tested; one PDA, one IBM laptop (older model) and a HP laptop (newer model).

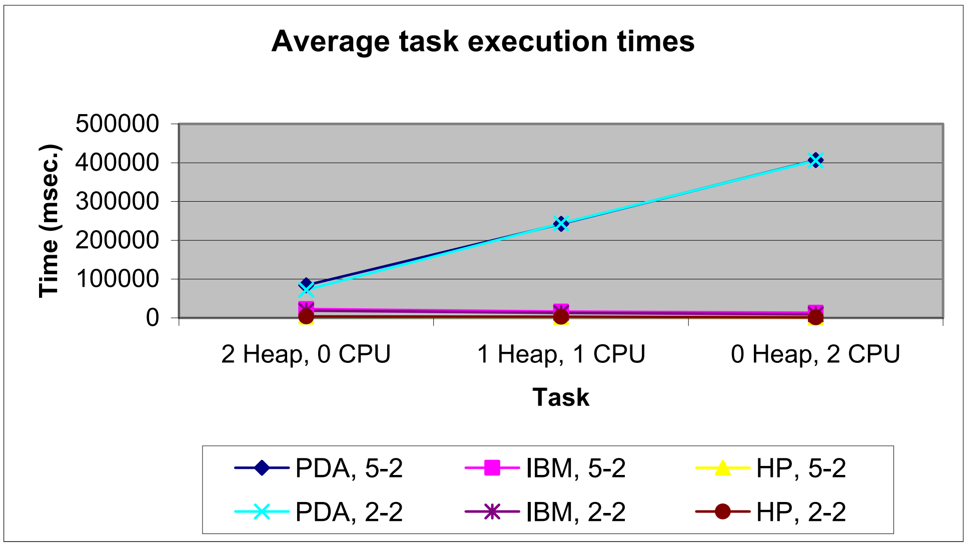


Figure 6-2: Summary of performance test-results

Some performance testing has also been carried out to determine the suitability for the proposed MicroTAPAS support in different network configurations with regard to Delay, Response time, overhead traffic-exact procedure and results have been reported as part of a submitted project report and will be continued within the same TAPAS project by another student.

6.5 Unresolved issues

A few unresolved issues were encountered during the testing of the system.

6.5.1 Available terminals

In order to perform an extensive test of the extended mobility functionality, a number of mobile terminals should have been employed, and for a duration of several hours, or indeed, days. One mobile terminal (PDA) was, however, the only device available for testing the system with, and a comprehensive test was thus not feasible.

6.5.2 Configuration of subnets

When attaching the two wireless LAN access points (AP) the university network, the networks DHCP server(s) handled the assignment and lease period of the addresses. The lease period, as mentioned in section 6.2, was set to 24 hours, and to simulate a change of domains, the address on the device had to be changed manually to simulate this transfer.

7 Faced challenges and possible solutions

Throughout this project, there have been a few unsuspected challenges and issues that had to be resolved in order to complete the assigned tasks. These challenges are described in this chapter, along with the solutions engineered along the way.

7.1 Connection time-out

When verifying a connection to the network, the ping client tries to establish a connection to a ping server on another node in the network, if a successful connection is established, the terminal/node has a network connection. This implemented feature is discussed in section 4.2.4.

A problem surfaced when a terminals ping client failed to established connection to a ping server, which indicates a loss of network connection. Due to the implemented Java Virtual Machines (VM) the ping clients attempt is not immediately published, and it will try to connect several times before the operation fails. This, on an average, takes around 50 seconds. In practise this means that from a network connection is lost, it can take as much as a minute before the terminal's TAPAS software is notified of the loss.

No real alternative or solution has been engineered to overcome this challenge, as it is due to the implementation of the VM.

7.2 Subnet masks & MAC addresses

It would be very useful to read subnet masks and the medium access control (MAC) addresses of a terminal, for use in addressing/routing and determining a terminal's domain. I.e. different domains could be assigned different subnet masks, independent of the actual IP address range of the various terminals connected to the domain, this would then simply routing of messages etc.

Due to limitations in Java, it has not been possible to determine which subnet mask an IP address has; as such functionality would be implemented on a lower level than Java normally operates on.

A workaround has been engineered where different domains are assigned a range of IP addresses. If a terminal is given a certain address, it then belongs to the domain which controls this address. This solution is working well, although it is more complex than a solution built on distinguishing between different subnet masks.

The same technical limitation applies to reading a network adapter's MAC address.

The MAC address is a unique address, identifying each network adapter, and could be used to uniquely identify each TAPAS node, for purposes of routing forwarded messages. In the original TAPAS architecture, each node is identified by the node part of the Global Actor Identifier (GAI), refer to Appendix A.1.3 for an overview of GAI, where the node part is the host name/IP address of the terminal. If an actor is moved, or a terminal is given a new address, this GAI is no longer applicable. The MAC address could somehow be used in this process to help in identifying moved actors and terminals.

A workaround has been engineered to overcome this, by keeping constantly updated tables of all the relevant information. The solution is working, but, as is the case with subnet masks, is more complex than it should be in theory.

8 Suggested improvements and further work

During the work with the TAPAS architecture in general and the extended support functionality for MicroTAPAS in particular, a few issues and ideas have come up. These are presented in this chapter.

8.1 Addressing

It was noted in section 7.2 that the GAI addressing scheme may have some shortcomings when implementing some of the mobility extensions to TAPAS. In particular TerminalMobility and ActorMobility suffer from this. If a different approach could be implemented, based on the uniqueness of, for example, MAC addresses, one could end up with an addressing scheme more suitable for further extensions to the support architecture. In particular, it would be suitable to implement a scheme where one could assign addresses independent of location.

8.2 Suitability of J2ME

In recent months there has been massive public criticism against Sun Microsystems Inc. for their response to a public demand to releasing a compiled Virtual Machine for terminals powered by Microsoft's PocketPC operating system. On the official email and discussion list (J2ME-CDC-INTEREST@JAVA.SUN.COM) several dozens of users/developers have argued that Sun should release their internal VM to the public (either for free or for a relatively low price) to make it easier, and more accessible, to develop J2ME applications for deployment to PDA's. A number of these requests point out that Sun is loosing territory against Microsoft's C# .NET programming language/developer platform for applications targeting handheld devices.

8.3 Session and user mobility

Session and User mobility should in a future version of MicroTAPAS be implemented, alongside the Terminal and Actor mobility functionality suggested by this report. An excellent proposal for such an extension is put forward by [LILL] in his report. That solution was originally specified and developed for the standard version of TAPAS, but could easily be implemented in MicroTAPAS as well.

8.4 Security

Different levels of encryption of all communication could be introduced to the MicroTAPAS architecture. The user, or the application, could be given the choice as to whether messages sent between MicroTAPAS instances should be encrypted, and to what degree, i.e. bit-length of keys and encryption algorithms. There are a lot of possibilities here that should be investigated, and could probably warrant a separate project in itself.

8.5 Reliability

Presently, there are no specific features built into the support system to ensure its reliability. This is one area that clearly should be addressed in the future, as the system is highly vulnerable to exceptions, failures and down-time of the web-server, the Director/MM nodes, or the underlying network itself.

8.6 Interoperability with TAPAS

A future version of MicroTAPAS should incorporate a model how to accomplish interoperability with the standard TAPAS support architecture. This task, however gets increasingly complex as more and more features are incorporated into the support architecture.

8.7 Applications

There are several applications that could be developed to take advantage of MicroTAPAS running on PDA's. Here are two examples described.

A Simple Network Management Protocol (SNMP) monitoring application is one example of a useful application, where a server maintains a list of nodes and status' that can be viewed by MicroTAPAS clients.

Another possible application is a mobile dynamic measurement of WLAN coverage and signal strength. Results could be sent back to server and made available to all other connected clients. It would also be possible to link this information to a map of a given area, and thereby building an easy to read coverage map for purpose of verifying WLAN coverage in a particular building or a campus.

9 Conclusion

The purpose of this project was to specify and implement mobility support for the MicroTAPAS support architecture. More specifically, actor and terminal mobility functionality was to be incorporated into the prototype architecture. In addition, a test application was to be developed and used for testing the implemented functionality.

TAPAS Extended Management (TXM) functionality was designed and implemented to provide extended mobility management functionality. This was realised through the MobilityManager actor entity. TAPAS Extended Support (TXS) functionality was designed and implemented to provide all needed behaviour to the generic but movable actors, as well as mobility support for mobile nodes and terminals. This was realised through the MobilityAgent actor entity. The resulting prototype underwent some testing, but less than was hoped for at the outset of the project. This was mainly due to three factors; the technical limitations posed by the current available J2ME runtime implementations, the relatively limited equipment available for use in testing and because of time constraints.

A test application, MicroTester v.2, was developed to take full advantage of the mobility extensions of MicroTAPAS, and was used during the testing of the system. A limited test was performed, as well as a successful demonstration for faculty and students at the institute of Telematics.

This report shows that TAPAS, in general, and MicroTAPAS in particular, is well suited for designing and implementing mobility support functionality to enable extensions to the original architecture. More work is, however, needed to further develop the prototype TAPAS support system.

Acronyms

This is a collection of the most used acronyms in this report.

AP	Access Point
APIR	ActorPlugInReq(uest)
DHCP	Dynamic Host Configuration Protocol
GUI	Graphical User Interface
J2ME	Java2 Micro Edition
MA	Mobility Agent
MAC	Medium Access Control
MM	Mobility Manager
PDA	Personal Digital Assistant
QoS	Quality of Service
SNMP	Simple Network Management Protocol
TAPAS	Telematics Architecture for Plug-and-Play Systems
TAS	TAPAS Actor Support
TCI	TAPAS Communication Infrastructure
TNES	TAPAS Node Execution Support
TXM	TAPAS Extended Management
TXS	TAPAS Extended Support

Bibliography

[AAGF1] Finn Arve Aagesen, Bjarne Helvik, Ulrik Johansen and Hein Meling, "*Plug and Play for telecommunication functionality – architecture and demonstration issues*", The International Conference on Information Technology for the New Millennium (IConIT2001), Thammasat University, Bangkok - Thailand, May 2001.

[AAGF2] Finn Arve Aagesen, Chutiporn Anutariya, Mazen Malek Shiaa and Bjarne E. Helvik, "*Capability Specification and Selection in TAPAS*", IFIP WG6.7 Workshop and Eunice Summer School on Adaptable Networks and Teleservices, Trondheim - Norway, September 2002.

[AAGF3] Finn Arve Aagesen, Bjarne E. Helvik, Chutiporn Anutariya, and Mazen Malek Shiaa, "*On Adaptable Networking*", The 2003 International Conference on Information and Communication Technologies (ICT 2003), Bangkok- Thailand, April 2003.

[COLU] Colombia University, Department of Computer Science, NetScript, http://www.cs.columbia.edu/dcc/netscript/. [Accessed October 2003]

[COUG] G. Coulouris, J. Dollimore, T. Kindberg, "Distributed systems, concepts and design", Third edition, Addison-Wesley, 2001.

[DARP] U.S. Department of Defence, Advanced Technology Office, http://www.darpa.mil/ato/programs/activenetworks/actnet.htm. [Accessed October 2003]

[FESA] A. Festag, "Mobility Support in IP Cellular Networks - A Multicast-Based Approach", PhD. Thesis, Fakultät IV - Elektrotechnik und Informatik der Technischen Universität Berlin, June 2003, [http://edocs.tu-berlin.de/diss/2003/festag_andreas.pdf]

[JOHU1] Ulrik Johansen, "*Dynamic Plug and Play - What is it, what are the advantages of using it?*" presented at IT-PRO 2000, Sandefjord, Norway.

[JOHU2] Ulrik Johansen, Finn Arve Aagesen, Bjarne E. Helvik and Rolv Bræk, "Design Specification of the PaP Support Functionality", Plug-and-Play Technical Report 2/1999, Department of Telematics, NTNU, 1999-12-10, ISSN 1500-3868

[LILL] Lars Erik Liljebäck, "User and Session mobility on a Plug-and-Play Architecture", MSc thesis, Department of Telematics, NTNU, 2002.

[LILL] Lars Erik Liljeback, “User and Session Mobility in a Plug-and-Play architecture”, MSc. thesis, Department of Telematics, NTNU, 2002.

[LUHE] Eirik Lühr, “*TAPAS for wireless PDA*”, Project Report, Department of Telematics, NTNU, 2003.

[MALM1] Mazen Malek Shiaa and Finn Arve Aagesen, “*Mobility management in a Plug and Play architecture*”, IFIP TC6 Seventh International Conference on Intelligence in Networks, Saariselka, Finland, April 2003. Plubliched by Kluwer Academic Publishers.

[MALM2] Mazen Malek Shiaa and Lars Erik Liljeback, “*User and Session Mobility in a Plug-and-Play Network Architecture*”, IFIP WG6.7 Workshop and EUNICE Summer School on Adaptable Networks and Teleservices, Trondheim - Norway, September 2002.

[MALM3] Mazen Malek Shiaa and Finn Arve Aagesen, “*Architectural Considerations for Personal Mobility In the Wireless Internet*”, Personal Wireless Communication (PWC2002), Singapore, October 2002.

[MALM4] Mazen Malek Shiaa, “*Mobility Support Framework in Adaptable Service Architecture*”, Network Control and Engineering for QoS, Security and Mobility 2003 IFIP/IEEE Conference (NetCon'2003), Muscat-Oman, October 2003.

[MELH1] Hein Meling, “Complete System Overview”, http://tapas.item.ntnu.no/documentation/SystemDoc/Main/Main.pdf, [Accessed October 2003]

[MITE] Massachusetts Institute of Technology, Active Networks, http://www.sds.lcs.mit.edu/activeware/. [Accessed October 2003]

[OMG1] Object Management Group (OMG), http://www.omg.org, [Accessed October 2003]

[OMG2] Object Management Group (OMG), “CORBA FAQ”, http://www.omg.org/gettingstarted/corbafaq.htm, [Accessed October 2003]

[REID] Reilly, David, “Mobile Agents -Process migration and its implications”, http://www.davidreilly.com/topics/software_agents/mobile_agents/. [Accessed October 2003]

[RFC2002] C. Perkins, Ed., "IP Mobility Support", RFC 2002, October 1996. [http://www.ietf.org/rfc/rfc2002.txt]

[RFC3344] B. Patil, P. Roberts, "IP Mobility Support for IPv4", RFC 3344, August 2002. [http://www.ietf.org/rfc/rfc3344.txt]

[STAU] Stanford University, Department of Computer Science, Knowledge Sharing Effort, http://www-ksl.stanford.edu/knowledge-sharing/. [Accessed October 2003]

[SUN1] Sun Microsystems Inc., “Jini Network Technology”, http://wwws.sun.com/software/jini/, [Accessed October 2003]

[UMBC] University of Maryland, Baltimore County, Lab for Advanced Information Technology, KQML Web, http://www.cs.umbc.edu/kqml/. [Accessed October 2003]

[UPEN] University of Pennsylvania, Department of Computer and Information Science,
Bellcore, http://www.cis.upenn.edu/~switchware/. [Accessed October 2003]

Appendix A - MicroTAPAS

A.1 Main changes between MicroTAPAS and basic TAPAS

A.1.1 Layered design model

The MicroTAPAS layered design model is a slightly modified version of the same model in TAPAS [MELH1, page 18]. Please refer to [LUHE, page 8] for a complete discussion about this topic. The modified design model used by MicroTAPAS is shown in Figure A-1 and the original model is shown in Figure A-2.

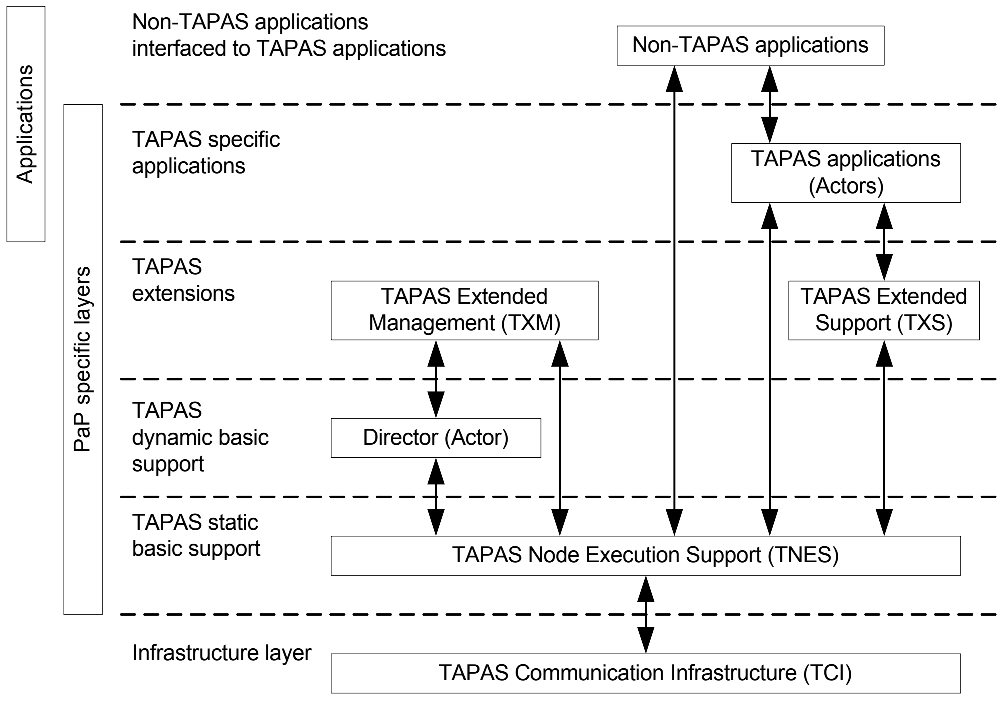

Applications
PaP specific layers
Non-TAPAS applications interfaced to TAPAS applications
Non-TAPAS applications
TAPAS specific applications
TAPAS applications (Actors)
TAPAS extensions
TAPAS Extended Management (TXM)
TAPAS Extended Support (TXS)
TAPAS dynamic basic support
Director (Actor)
TAPAS static basic support
TAPAS Node Execution Support (TNES)
Infrastructure layer
TAPAS Communication Infrastructure (TCI)

Figure A-1: MicroTAPAS layered design model - architecture

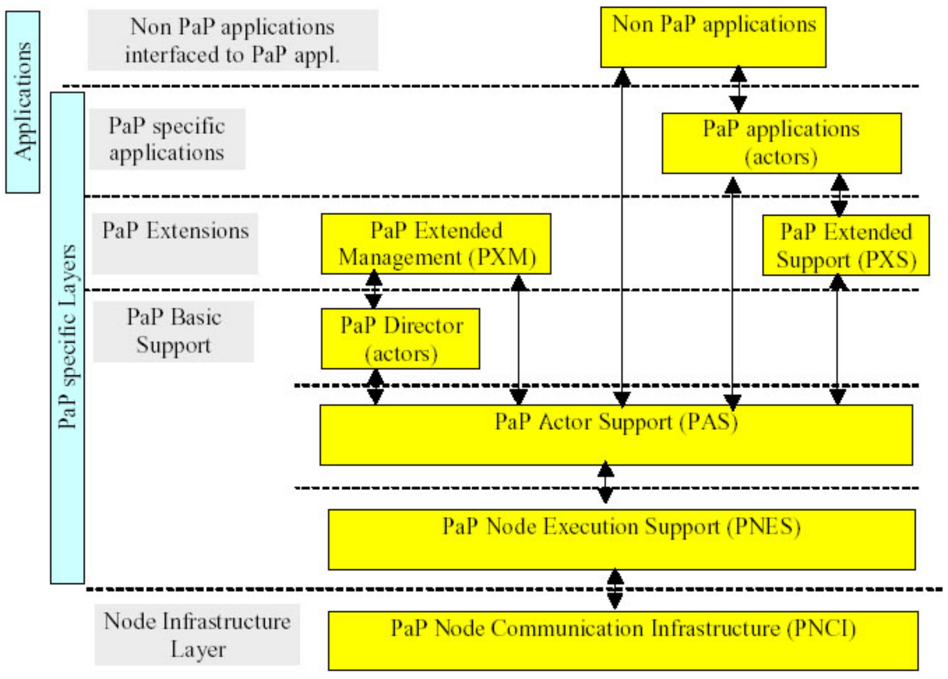

Applications
PaP specific Layers
Non PaP applications interfaced to PaP appl.
Non PaP applications
PaP specific applications
PaP applications (actors)
PaP Extensions
PaP Extended Management (PXM)
PaP Extended Support (PXS)
PaP Basic Support
PaP Director (actors)
PaP Actor Support (PAS)
PaP Node Execution Support (PNES)
Node Infrastructure Layer
PaP Node Communication Infrastructure (PNCI)

Figure A-2: TAPAS layered design model - architecture

A.1.2 Communication

The biggest difference between TAPAS and MicroTAPAS, in terms of how the communication is carried out, is the absence of RMI. It was decided to carry out all communication by using a combination of Java sockets and local method calls. Please refer to [LUHE, page 13] for a complete discussion of this topic.
Figure A-3 shows the new communication model, and Figure A-4 shows the communication model used in standard TAPAS.

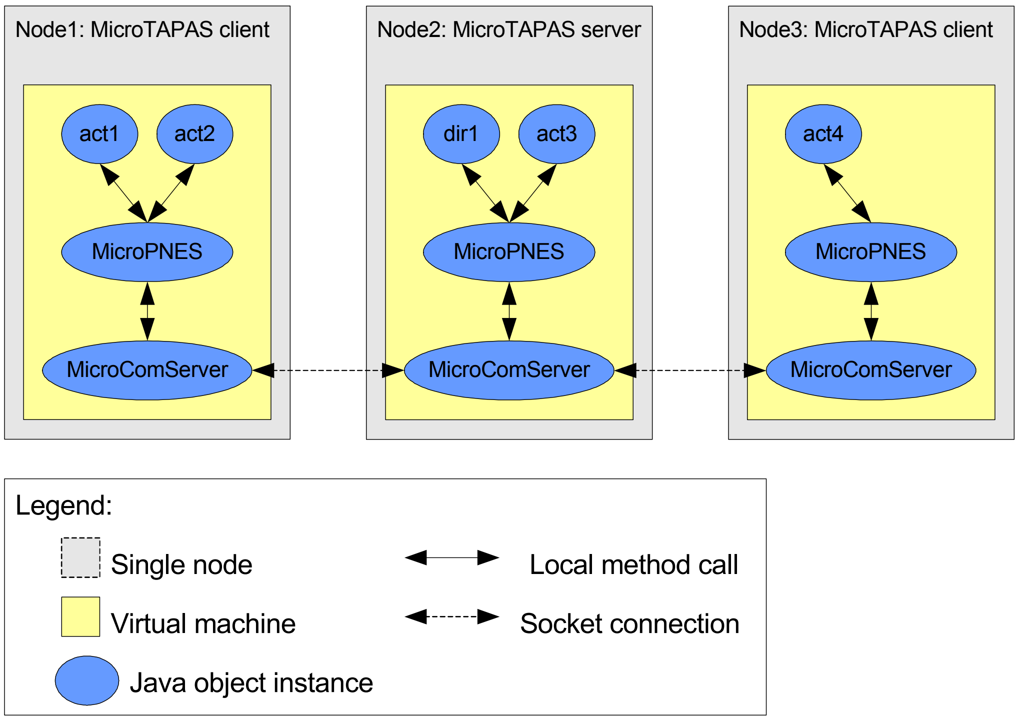


Figure A-3: MicroTAPAS synchronous communication model

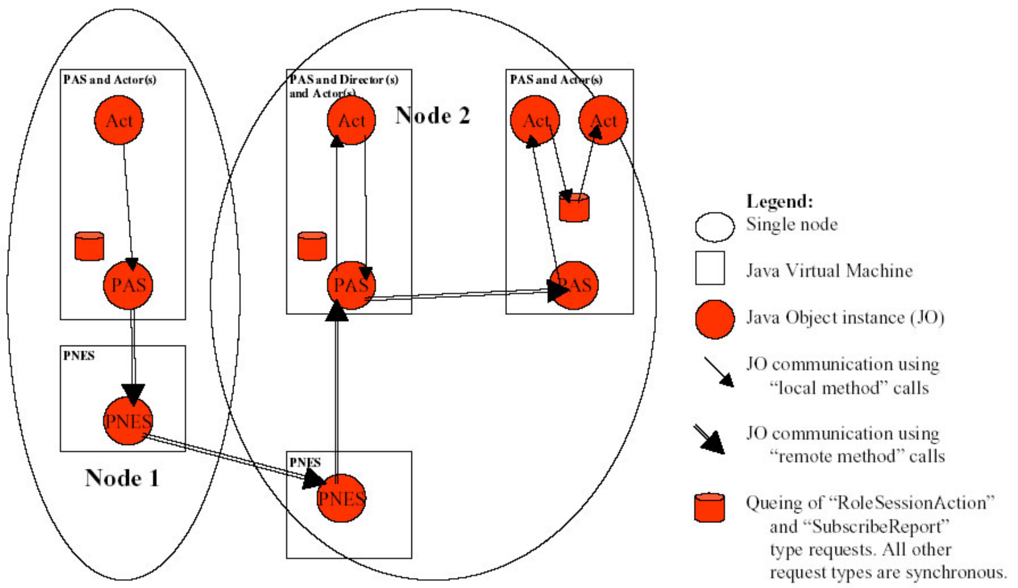

PAS and Actor(s)
PAS and Director(s) and Actor(s)
PAS and Actor(s)
Act
PAS
PNES
Node 1
Node 2
Legend:
Single node
Java Virtual Machine
Java Object instance (JO)
JO communication using "local method" calls
JO communication using "remote method" calls
Queing of "RoleSessionAction" and "SubscribeReport" type requests. All other request types are synchronous.

Figure A-4: TAPAS synchronous communication model

A.1.3 Addressing and routing

The addressing and routing in MicroTAPAS are based on the same principles as that of TAPAS, although the absence of the PAS layer required a slight modification. Figure A-5 shows the slightly modified type of Global Actor Identifier (GAI) used by MicroTAPAS. The original TAPAS GAI is shown in Figure A-6.

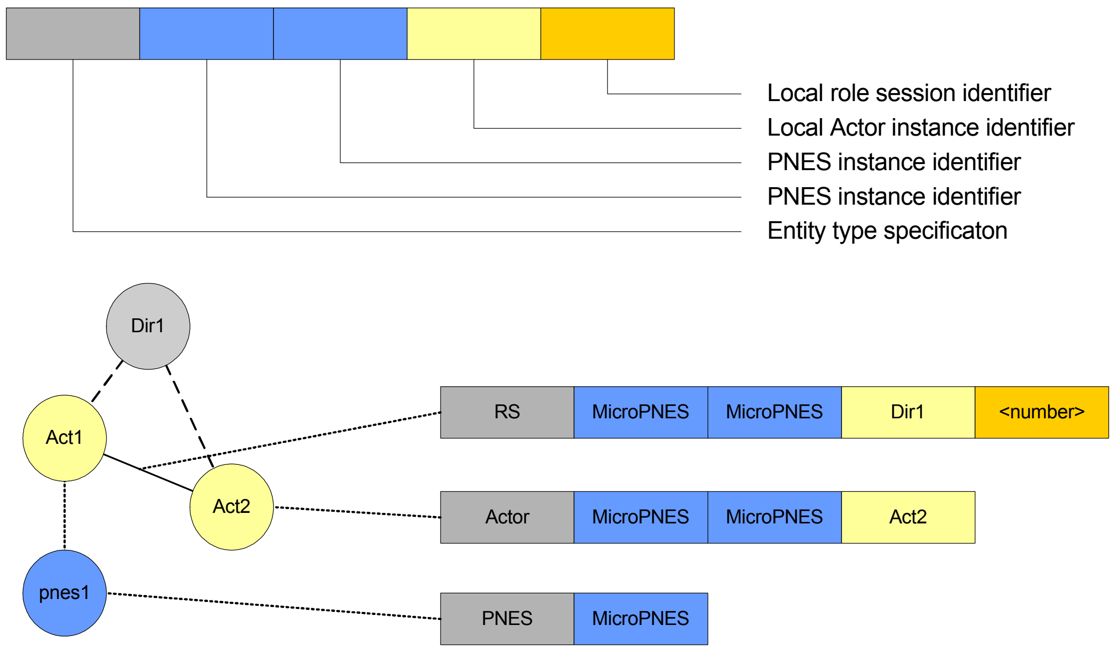


Figure A-5: MicroTAPAS addressing scheme

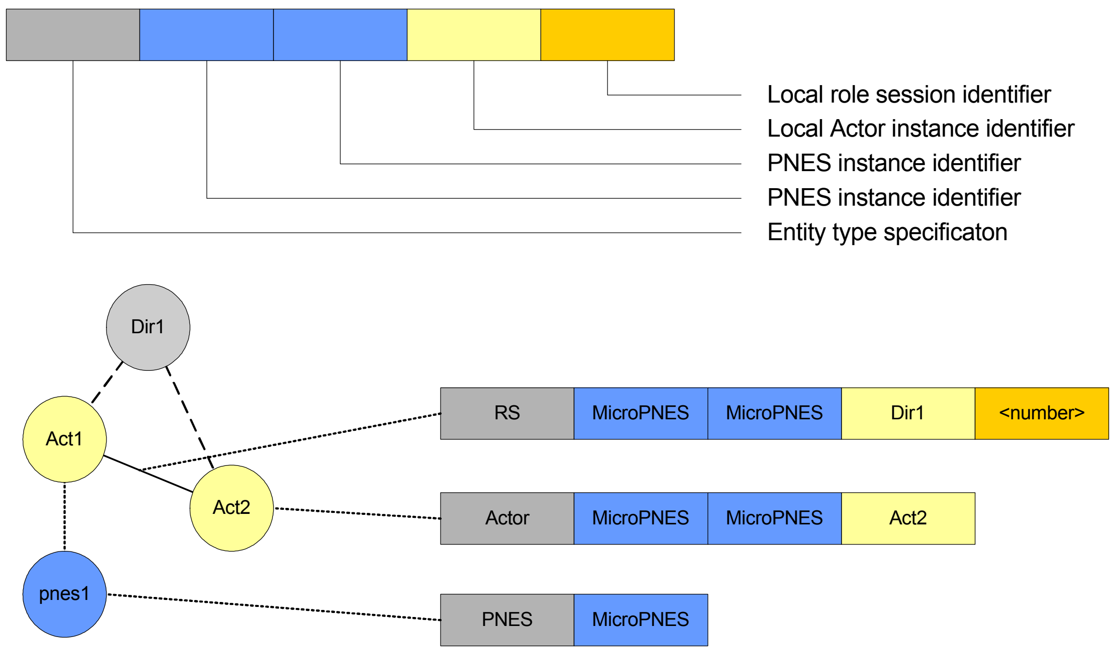


Figure A-6: TAPAS addressing scheme

A.2 Performance comparison between PDA and laptop

A.2.1 Hardware used

The following table show in detail the hardware used for testing the MicroTAPAS architecture.

	Node A	**Node B**	**Node C**
Classification	LAPTOP	PDA	LAPTOP
Model	IBM Thinkpad 390E	iPAQ H.3660	HP Pavillion ze5244
Processor	Intel Pentium II, 266 MHz	Intel StrongARM, 206 Mhz	Intel Pentium 4 M, 2,53 GHz
Memory (RAM/ROM)	192 MB/x	64 MB/16 MB	512 MB/x
Operating System (OS)	Microsoft Windows XP, Professional Edition	Microsoft PocketPC 2000	Microsoft Windows XP, Home Edition
Network	SMC	ZyAIR B-100, IEEE 802.11b compliance at 2.4 GHz, PCMCIA card	LAN-Express IEEE 802.11 PCI Adapter
Virtual Machine (VM)	Sun Microsystems Inc’s Java HotSpot(TM) Client VM, version 1.4.1_02-b06	IBM’s J9, version 2.0	Sun Microsystems Inc’s Java HotSpot(TM) Client VM, version 1.4.1_02-b06

A.2.2 Screen output from test-program execution

The following data was printed to screen during one of the test runs, and, as can be seen on the top, it was started with five threads and two cycles. All the times quoted are in milliseconds. The first column states the number of threads used to execute the tasks, the other three columns show the different tasks, as explained in [LUHE, page 37].

```
C:\dev\MicroTAPAS>java heaptest.HeapTest 5 2
Unable to open log file. Printing to System.out...

Max # threads =      5
Total (heap + CPU) cycles = 2


# Threads       2 Heap, 0 CPU   1 Heap, 1 CPU   0 Heap, 2 CPU

1       21981   20409   15161

2       20420   14901   15863

3       31085   17125   10254

4       20480   12858   11396

5       19718   14441   12718
```

A.2.3 Data from performance testing of laptop vs. PDA

This is the data values from the performance testing performed in [LUHE, page 37]. The top-left heading in each table states which terminal and start parameters were used. For example, IBM, 5-2 states that it was run on the IBM laptop (node A) with a maximum of five threads and two cycles. Figure 6-2 (page 44) was drawn by computing the average execution time for each task (i.e. the first value for 'IBM, 5-2' in the figure was found by summing the column '2 Heap, 0 CPU' and dividing by the number of threads; five).

IBM, 5-2	Threads	2 Heap, 0 CPU	1 Heap, 1 CPU	1 Heap, 1 CPU
	1	21981	20409	15161
	2	20420	14901	15863
	3	31085	17125	10254
	4	20480	12858	11396
	5	19718	14441	12718

HP, 5-2	Threads	2 Heap, 0 CPU	1 Heap, 1 CPU	1 Heap, 1 CPU
	1	4046	2844	1422
	2	3706	2613	1422
	3	3325	2364	1412
	4	3264	2434	1422
	5	3044	2173	1422

PDA, 5-2	Threads	2 Heap, 0 CPU	1 Heap, 1 CPU	1 Heap, 1 CPU
	1	64285	258435	419323
	2	80407	239302	403596
	3	87770	238447	403535
	4	93045	237556	404077
	5	93945	238113	404354

IBM, 2-2	Threads	2 Heap, 0 CPU	1 Heap, 1 CPU	1 Heap, 1 CPU

	1	19197	13920	9193
	2	17705	11467	9193

HP, 2-2	Threads	2 Heap, 0 CPU	1 Heap, 1 CPU	1 Heap, 1 CPU
	1	4096	2894	1432
	2	3605	2574	1412

PDA, 2-2	Threads	2 Heap, 0 CPU	1 Heap, 1 CPU	1 Heap, 1 CPU
	1	62779	246339	406098
	2	81344	241102	406206

Appendix B - MicroTAPAS with Mobility extension

B.1 Node

B.1.1 Example node configuration file

```
# Now it is safe to add comments, prefixing each line with a hash '#'. Empty
# lines and white space is also ok.

############################
#
# BASIC PARAMETERS
#
############################
# The Internet address of the binary files of the whole system.
# The address should be on the form:
#  http://<ip.addr or url>/<tapas directory>/
#codebase = http://129.241.200.218/tapasroot/
codebase = http://www.stud.ntnu.no/~luhr/tapasroot/

# The GAI (address) of the Director
#homeinterface = Actor://127.0.0.1/MicroPNES/MicroTAPAS.MicroDirector1
homeinterface = Actor://localhost/MicroPNES/MicroTAPAS.MicroDirector1

# The GAI of the local MicroPNES instance. 'localhost' will automatically be
# replaced by the nodes real IP address.
selfinterface = PNES://localhost/MicroPNES/MicroPNES

# Port used for socket communication between MicroTAPAS nodes (universal)
communicationport = 9999

# Location of the TAPAS bootstrap root directory
tapasroot = /dev/TAPAS/MicroTAPAS_thesis/MicroTAPASBoot/StartMicro/

# Location for storing data files locally (below the 'tapasroot' directory)
dataroot = data/

# File for storing the node capabilities
nodecap = node.cap

# File used for storing the script that should be run after startup
nodescript = node.script

# Determines whether the script should automatically run at startup of
MicroPNES
runscriptauto = false

############################
#
# MOBILITY PARAMETERS
#
############################
# The GAI of this domain's MobilityManager
mminterface =
Actor://localhost/MicroPNES/MicroTAPAS.mobility.MobilityManager1

# The GAI of the local MobilityAgent. 'localhost' will automatically be
# replaced by the nodes real IP address.
mainterface = Actor://localhost/MicroPNES/MicroTAPAS.mobility.MobilityAgent1

# Ping-server port (universal)
pingport = 9998

# Initial interval between pings, in milliseconds (dynamic)
```

```
pinginterval = 10000

# Interval between an actor checks for capability updates, in milliseconds
(static)
# A value of -1 will prevent the monitor from running.
capsmoninterval = 5000

############################
#
# DEBUG PARAMETERS
#
############################
# The location of the debugserver
#debugserver = 127.0.0.1:9990
debugserver = 129.241.200.218:9990

# Is debug info to be printed to screen?
debug = false

############################
#
# VISUAL PARAMETERS
#
############################
# Frame icons
icon.micropnes = /icons/greendot.gif
icon.microdirector1 = /icons/dirIcon.gif
icon.mobilitymanager1 = /icons/com.gif
icon.mobilityagent1 = /icons/com_red.gif
icon.debugserver = /icons/yinyangIcon.gif
icon.default = /icons/ntnuIcon.gif
```

B.2 MobilityApplicationActor

B.2.1 Actor configuration file

This is an example actor configuration file.

```
# Should the actor automatically register with its local MobilityManager?
autoregister = false

# Variable that decides what strategy to use when actor should be moved
# alternatives are: 'a', 'b' or 'c' ('a' is the most simple)
movestrategy = a

# Default location to move actor to, if offered capabilities detoriate
movelocation = PNES://10.0.0.1/MicroPNES/MicroTesterMoved
```

B.2.2 Actor capability file

This is an example actor capability file.

```
arhcitecture.*
cpu.clock.100
memory.ram.64
screen.colors.65000
screen.resolution.x.240
screen.resolution.y.320
```

Appendix C - Javadoc

To prevent this appendix to become very large, only the class hierarchies have been included for each of the three main parts of the implementation; the mobility package, the debug package and the tester package

C.1 Hierarchy for MicroTAPAS.mobility package

- class java.lang.Object
    - class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
        - class java.awt.Container
            - class java.awt.Window (implements javax.accessibility.Accessible)
                - class java.awt.Frame (implements java.awt.MenuContainer)
                    - class MicroTAPAS.mobility.**MobilityManagerFrame** (implements java.io.Serializable)
    - class MicroTAPAS.**MicroActor** (implements MicroTAPAS.ControlInterface, MicroTAPAS.MicroActorInterface, java.io.Serializable)
        - class MicroTAPAS.**MicroApplicationActor** (implements java.io.Serializable)
            - class MicroTAPAS.mobility.**MobilityApplicationActor** (implements java.io.Serializable)
        - class MicroTAPAS.mobility.MobilitySupportActor (implements MicroTAPAS.ControlInterface, MicroTAPAS.MicroActorInterface)
            - class MicroTAPAS.mobility.**MobilityAgent1**
            - class MicroTAPAS.mobility.**MobilityManager1**
    - class MicroTAPAS.mobility.**MobilityRegistryManager** (implements java.io.Serializable)
    - class MicroTAPAS.mobility.**MobilityRequest** (implements java.io.Serializable)
    - class MicroTAPAS.mobility.**Node** (implements java.io.Serializable)
    - class java.lang.Thread (implements java.lang.Runnable)
        - class MicroTAPAS.mobility.**CapsMonitorAgent** (implements java.io.Serializable)
        - class MicroTAPAS.mobility.**MicroPingClient** (implements java.io.Serializable)
        - class MicroTAPAS.mobility.**MicroPingServer** (implements java.io.Serializable)

C.2 Hierarchy For Package MicroTAPAS.debug

- class java.lang.Object

- class javax.swing.table.AbstractTableModel (implements java.io.Serializable, javax.swing.table.TableModel)
    - class MicroTAPAS.debug.**DebugTableModel**
- class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - class java.awt.Container
        - class javax.swing.JComponent (implements java.io.Serializable)
            - class javax.swing.JTable (implements javax.accessibility.Accessible, javax.swing.event.CellEditorListener, javax.swing.event.ListSelectionListener, javax.swing.Scrollable, javax.swing.event.TableColumnModelListener, javax.swing.event.TableModelListener)
                - class MicroTAPAS.debug.**DebugTable**
        - class java.awt.Window (implements javax.accessibility.Accessible)
            - class java.awt.Frame (implements java.awt.MenuContainer)
                - class javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
                    - class MicroTAPAS.debug.DebugBaseFrame2 (implements java.io.Serializable, java.awt.event.WindowListener)
                        - class MicroTAPAS.debug.**DebugFrame2** (implements java.awt.event.ActionListener, java.awt.event.ItemListener, javax.swing.event.TableModelListener)
- class MicroTAPAS.debug.**Debug** (implements java.lang.Runnable)
- class MicroTAPAS.debug.**DebugEvent** (implements java.io.Serializable)
- class MicroTAPAS.debug.**DebugServer** (implements java.io.Serializable)

C.3 Hierarchy For Package MicroTester.v2_0

- class java.lang.Object

- class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
  - class java.awt.Container
    - class java.awt.Window (implements javax.accessibility.Accessible)
      - class java.awt.Frame (implements java.awt.MenuContainer)
        - class MicroTester.v2_0.**TesterDialog1** (implements java.awt.event.ActionListener, java.awt.event.KeyListener)
        - class MicroTester.v2_0.**TesterDialog2** (implements java.awt.event.ActionListener, java.awt.event.KeyListener)
        - class MicroTester.v2_0.**TesterMainWindow** (implements java.awt.event.ActionListener)
- class MicroTAPAS.**MicroActor** (implements MicroTAPAS.ControlInterface, MicroTAPAS.MicroActorInterface, java.io.Serializable)
  - class MicroTAPAS.**MicroApplicationActor** (implements java.io.Serializable)
    - class MicroTAPAS.mobility.**MobilityApplicationActor** (implements java.io.Serializable)
      - class MicroTester.v2_0.MicroTesterBase (implements java.io.Serializable)
        - class MicroTester.v2_0.**MicroTester1**
        - class MicroTester.v2_0.**MicroTester2**
      - class MicroTester.v2_0.**MicroTesterServer**
- class MicroTester.v2_0.**ServerInterface** (implements java.io.Serializable)
- class MicroTester.v2_0.**TesterInterface** (implements java.io.Serializable)