OPEN Project

STREP Project FP7-ICT-2007-1 N.216552

Title of Document:	Final OPEN Service Platform architectural framework
Editor(s):	Susan Thomas
Affiliation(s):	SAP AG
Contributor(s):	All Open Partners
Affiliation(s):	All Open Partners
Date of Document:	August 26, 2009
OPEN Document:	D1.4
Distribution:	EU
Keyword List:	
Version:	1.0 (Final Version)

OPEN Partners:

CNR-ISTI (Italy)
Aalborg University (Denmark)
Arcadia Design (Italy)
NEC (United Kingdom)
SAP AG (Germany)
Vodafone Omnitel NV (Italy)
Clausthal University (Germany)

ABSTRACT

The purpose of this deliverable is to present the final OPEN Service Platform architectural framework. It details the main phases of UI migration and application logic reconfiguration, and also discusses the advantages and disadvantages of various design options. It describes the overall architecture of the Migration Service Platform (MSP), and provides a high level description of all of the platform components and the interfaces between them. In this way, it creates a common vocabulary and understanding, which will guide the integration and implementation work still to be done in WP4.

TABLE OF CONTENTS

1. INTRODUCTION

This document presents a refinement of the architectural framework of the Migration Service Platform (MSP), which was developed in the first year of the project, and which was described in D1.2. It retains the main concepts and phases of the migration process as well as the concepts and processes for application logic reconfiguration, as these were incorporated into prototypes, which confirmed their viability.

This final version of the overall architecture reflects our increasing understanding of the issues of migratory applications, which resulted from work that was done, and which is documented in D1.3 ("Final Requirements for OPEN Service Platform"), D2.1 ("Early infrastructure for migratory interfaces"), D2.2 ("Document about architecture for migratory user interfaces "), D3.1 ("Detailed Network Architecture"), D4.1 ("Solutions for application logic reconfiguration"), D4.2 ("Migration service platform design") and D5.1 ("Initial Application Requirements and Design").

The organization of the document is as follows. Sections 2, 3 and 4 introduce the major concepts related to migratory applications. Section 5 introduces the architecture for a platform to support migratory applications. Sections 6 and 7 describe two major capabilities of migratory applications: user interface migration and application logic reconfiguration, and also describes how these two functions are supported by components of the platform. Section 8 gives an overview of all of the platform components that support migratory applications. Section 9 presents two scenarios of migratory applications, adding some details about how platform components interact with each other and with applications/users. Finally, Section 10 is the conclusion.

2. MIGRATORY APPLICATIONS

One important aspect of ubiquitous environments is to provide users with the possibility to freely move about and continue to interact with the available applications through a variety of interactive devices such as cell phones, PDAs, desktop computers, digital television sets or intelligent watches. In such environments one potential source of significant frustration is that people have to start their application session over again from the beginning after changing to a different interactive device.

Migratory applications can overcome this limitation. Migratory applications, as defined by OPEN, are applications which are able to follow users, sense the users' context (where context is any information that can be used to characterize the situation of an entity [Dey00]), and adapt to the changing context, e.g., set of available devices, while also preserving the continuity of application sessions, thereby ensuring the continuity of the tasks supported by the application.

To summarize:

Migration = Device Change + Adaptation + Continuity.

As we will describe in the following sections, in order to have a real 'migration', all such aspects have to be included: no proper 'migration' occurs if there is a change of device and an adaptation of the application features to the new device, but there is no continuity in the resulting user activity because, for instance, the user has to restart from the beginning when the new configuration is activated. Likewise, a situation in which there has been a device change, and also the state of the application has been preserved, cannot be properly called 'migration', if adaptation is called for, but is not performed.

Therefore, migration encompasses all three aspects and the related issues: *device change*, where there is the issue of how devices are *discovered* and selected; *adaptation*, where there is the problem of how the characteristics of the context are taken into account and handled when adapting the application to the characteristic of the new context; and *continuity*, where there is the issue of how to guarantee continuity in task performance, and further issues like the necessity of techniques for *preserving the state*.

Thus, the OPEN project provides integrated solutions able to address all three aspects: device change, task continuity and adaptation. This is obtained through the Migration Service Platform (MSP), a middleware for migratory applications. With a migratory application, users can change which interactive devices and which networks are used to interact with the application, can have the interaction adapted to the form and features of the new set of devices, and can seamlessly continue their work, using their existing sessions with the application.

Migration is made possible by the MSP and the underlying networks, and may also involve reconfiguration, re-wiring, replacement or migration of components of the application logic. Figure 1 presents a high-level view of the intent of the OPEN Migration Service Platform, which is to enable software developers to turn an application into a migratory application.

Ideally, the migration platform should be able to take all existing applications and make them migratory. However, this is clearly too ambitious for a single project, thus we focus on classes of

applications, in particular, Web applications and distributed applications in the game and business domains. In addition, the project looks at the extent to which such applications should be developed according to some guidelines in order to ease the support given by the migration platform.

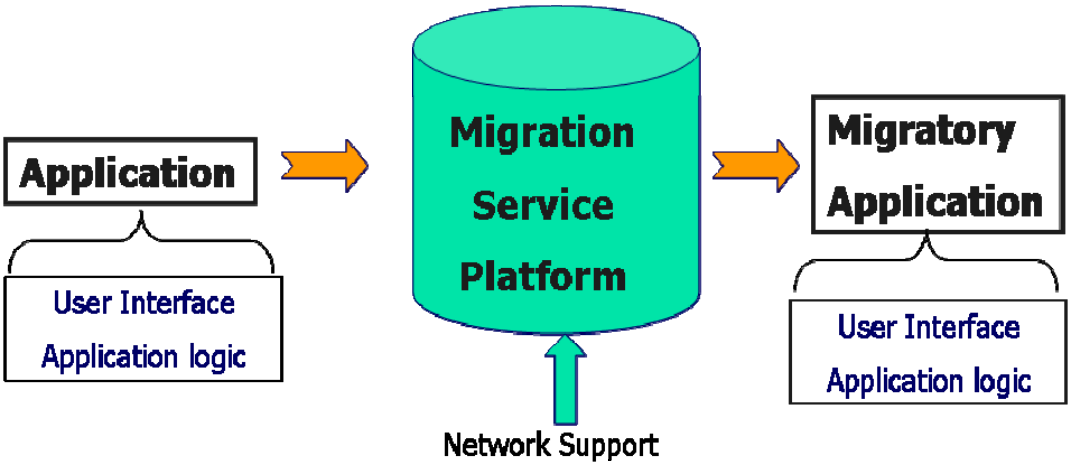

Figure 1 Migratory Application Supported by MSP and Network

3. CAPABILITIES OF A MIGRATORY APPLICATION

As indicated in Figure 1 an application consists of two basic types of components which are relevant to OPEN: user interface (UI) components and application logic components. Both of these can be migrated and adapted by a migratory application with the aid of the OPEN MSP. In addition, a migratory application can adapt to changes in the availability or quality of the networks in the current environment of the user. Thus, as users move about or as their environment changes, a migratory application can perform three basic types of actions, which can be done singly or together.

- *Migration of the user interface*: This type of migration allows users to change the interaction device and to continue their current tasks. User interface migration requires the system to capture the state of both the user interface and the application logic in order to send it to the new device (transformed if necessary) to continue interaction. The MSP will also allow for partial migration of a user interface. Partial migration enables users to switch from single device interaction to multiple device interaction or vice versa. An example of partial migration from the OPEN social game scenario is moving the input control from a PC keyboard to a mobile phone, while leaving the presentation of the output at the PC. UI Migration also involves adaptation of interaction to the form and capabilities of the target device, e.g., a switch to more computationally intensive graphical special effects, if the device has enough cores.

- *Application Logic Reconfiguration*: This concerns the reconfiguration, and sometimes the migration, of the logical (computational) components of the application. Whenever the new device or new devices provide a new set of features or functions which are not supported by the current devices, a migration and reconfiguration of the application logic may be required. A reconfiguration often requires a change in the set of currently active application components. Some new components might be needed; others might be not needed anymore.

- *Network Reconfiguration*: Network availability, connectivity and quality can vary from place to place, from device to device, and from time to time. Migratory applications can react to such variation by switching which networks are used by which devices, or by re-configuring how the networks are used by the devices. If the networks support it, migratory applications can also simultaneously switch to a different device and a different network without interruption to their sessions, e.g. audio or video sessions. From the user's point of view, this switch looks like both a network switch and a user interface migration with task continuity.

3.1. ADAPTATION - WHAT CAN BE ADAPTED?

Adaptation can take place at various levels of granularity. For example, Aura [GSS+02] provides support for migration, but the solution adopted has a different granularity. In Aura, for each possible application service, various applications are available and the choice of the application depends on the interaction resources available. Thus, for example for word processing, if a desktop is available then an application such as MS-Word can be activated, whereas in the case

of a mobile platform a lighter editing application is used. Thus, Aura aims to provide adaptation, but this is obtained mainly by changing the application depending on the resources available in the device in question.

By contrast, the MSP enables the development of migratory applications which adapt themselves to the user's context, including the available interaction resources. It is not just a choice among applications like Aura. In fact, various parts of an interactive migratory application can be adapted, including: the user interface, the application logic, the application content and the use of underlying networks. Each of these is considered in turn.

The user interface is composed of the presentation (the choice of the modality, layout, graphical attributes, etc.), the dynamic behavior (the choice of the navigation model, the dynamic activation and deactivation of interaction techniques), and content (what information is actually presented). Each of these can adapt in response to changes in context. Adaptation can be performed according to various strategies such as:

- Conserving (keep the arrangement/presentation of UI objects)
- Rearrangement (UI objects kept during migration but they are rearranged according to some techniques, e.g.: different layout)
- Increase (target device can provide more UI features)
- Reduction (less UI features)
- Simplification (UI objects kept, but with simplified representations, e.g.: images with lower resolutions)
- Magnification (opposite of simplification).

The application logic can be adapted by reconfiguring the access to the functionalities in order to access different implementations of some of them or increase/decrease such functionalities because of the change of device or the change of the connectivity. For example, an access to a database to retrieve a large amount of data can be performed, if the application has good network connectivity, while the same access should be avoided if the connectivity is too poor to provide results in a reasonable amount of time. In this case, it could be necessary to perform the database access in a separate phase or in a separate application, hence affecting not only the business logic of the migrating application but also the overall procedure the migrating application belongs to.

The content, which is stored in the functional core of the application, can also be adapted. There are two possible alternatives: one possibility is that different representations of the same content can be statically maintained and, depending on, e.g., the resources of the device at hand, the most suitable content will be selected. Another possibility is that the adapted content can be dynamically generated from the content that is already available. For instance, this is the case, when an image can be derived from a video, by capturing just the first frame.

Finally, a migratory application can also adapt its use of networks. Since connectivity can change, the networks used by the application and their quality of service may have to change. We should emphasize that network aspects can cover two different issues. On the one hand, the network can be adapted in order to offer better Quality of Service (QoS). In this case, the network properties will be adjusted in order to fulfill some requirements. In other cases, the network itself can be considered as an additional contextual aspect (e.g.: it can be considered as an aspect of the physical environment): depending on the current network capability, other aspects of the system (e.g.: the user interface presented on device, etc.) can be adapted.

3.2. THE MIGRATION PROCESS

The migration process, for either UI or logical components, can be seen as composed of a number of phases, which can be implemented in various ways. Such phases are specific functions which characterize the migration process, and which the migration platform should explicitly support:

- *Device Discovery*. Its purpose is to identify the devices that are available to be involved in the migration process and their attributes that can be relevant for migration (private or public device, their connectivity, their interaction resources, etc.).

- *When to Migrate*. A *migration trigger* indicates when to migrate. This event can be generated by the user or the system, or through a mixed initiative process (the system proposes migration and the user can decide whether to accept it). Users can request migration when they feel it necessary, while the system can trigger it when specific events are detected (such as the device is running out of power).

- *Where to Migrate*. Once migration is triggered, it is important to identify the target device for the migration process. Such a target should be one of the devices available for this purpose, and it should be selected on the basis of its features and how well it fits in the new context of use. There may be some interaction between *where to migrate* and *what to migrate*.

- *What to Migrate*. An interactive migratory service is composed of two main types of components: user interface and application logic components. The former is the software dedicated to the interaction with the user, while the latter is the functional core, independent of how user interaction takes place. A migratory application gives users appropriate control over which components to migrate. For example, the user might choose to change their input device from a keyboard to a motion sensor, while leaving the output at the source device.

- *How to Migrate*. Since the device to access the application usually changes after migration, some level of adaptation of the migratory application should be performed, in particular of its interactive part, in order to better exploit the new resources available while preserving usability.

- *Task Continuity*. One of the main reasons for migration is to continue a session and on-going tasks through different devices. This means that the work done by the user with the source device should not be lost when moving to the target device. Thus, it is necessary to extract the state of components at the source device and to transfer that state to the components at the target device.

- *Activation for the target*. In order to obtain continuity, it is important that the application is activated not at its usual starting point, but at the point at which it left off on the source device. This applies to streaming media as well.

- *Optional termination of the source*. In general, after migration, use of the source device should probably be terminated, but there are cases in which it can be useful to allow access to the migratory application from both the source and the target device after the migration.

4. MIGRATION LOGICAL DIMENSIONS

4.1. MIGRATION TYPES

Various types of application migration can be identified, depending on the number of source and target devices involved:

Total – all interaction with the application is moved from one source device to one target device.

Partial – some elements of the interaction with the application move from the source device to a target device. For example, input may move from a keyboard to an accelerometer.

Distributing – a migration where different interactive parts on one device move to different devices. This can also be seen as a collection of partial migrations.

Aggregating – a migration which involves multiple source devices and one target device.

Multiple – a migration which involves multiple sources and multiple targets.

In general, *migration* may mean a replication or a move, where *replication* means interactivity using the source device is still available, and *move* means it is not.

Figure 2 demonstrates a partial migration in which the application output is moved from the pocket PC to a local PC monitor, but the input controls remain on the pocket PC, and are adapted to utilize the space freed by moving the output.

Figure 2 Example of Partial Migration

4.2. TRIGGER TYPES, POLICIES AND TARGET DEVICE SELECTION

The trigger event defines when migration should occur. It should then be accompanied by the indication of the target device(s) available for migration. In order to allow for a good choice of the target device, the migration platform should retrieve and store information about the devices that are automatically discovered in the environment. The collected information mainly concerns device identification and their capabilities. On the one hand, such information allows users to choose a target migration device with more accurate and coherent information on the available target devices and, on the other hand, it allows the system to suggest or automatically trigger, subject to user specified policies, migrations when the conditions for migration arise. Thus, both the system and the user have the possibility to trigger the migration process, depending on the surrounding context conditions and policies set by the user. Users can have various ways of issuing migration requests. One example is to graphically select the desired target device from a list of devices. Users should have the possibility of choosing those devices that they are allowed to use, and that are currently available for migration. (See D3.1 for more information; also see Figure 6 and Figure 7 for diagrams of user and system initiated triggering, respectively)

Migration can also be initiated by the system without explicit user intervention in critical situations when the user session could accidentally be interrupted by external factors. For

example, we can foresee the likelihood of having a user interacting with a mobile device that is shutting down because its battery power is getting too low. Such situations can be recognized by the system and a migration can be automatically started to allow the user to continue the task from a different device, avoiding potential data loss. Alternatively, the migration platform can provide users with migration suggestions in order to improve the overall user experience. This happens when the system detects that in the current environment there are other devices that can better support the task being performed by the user. For example, if the user is watching a video on a PDA and a wide wall-mounted screen is detected and available in the same room, the system will prompt the user to migrate to that device, as it could improve her/his performance. However, the user can continue to work with the current device and refuse the migration. Since receiving undesired migration suggestions can be annoying for the user, users who want to receive such suggestions when better devices are available must explicitly formulate a policy allowing this sort of mixed-initiative migration service. Such a policy might specify, e.g., that the system should only issue a trigger if it is very certain that the migration will not fail in the current context, where context might include factors like the network connectivity and the speed of the user's movement.

In any case, once a migration has taken place, nothing prevents the user or the system from performing a new migration to another available device.

4.3. CONTEXT

Context information refers to all information that is relevant to describe a given situation for a given object, and as a general definition we use the one given by [Dey00]:

> ***Context*** *is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.* [Dey00]

Knowledge of context is useful to ensure that migration occurs at the right time and place.

The definition of context is broad, and implies that context may be any general type of information. The key is, as indicated, that it is *relevant* information that falls under the definition of context. Thus in OPEN we need first to define what is relevant information for service migration. Various parts of the context of use can be relevant in the migration process. A change in a single contextual dimension (which, in turn, results in a more general *context change*) might represent a condition to be analyzed for activating a possible migration. The context of use can be seen as composed of four main aspects:

- *Device*, its resources for interaction and processing, and its network connectivity. A device change is a common example of a context change. A device change can affect different aspects in a migrating application, not only from a user interface point of view (because the new device might offer different interaction capabilities with respect to the old one), but also e.g. from a performance point of view (the new device can support

more concurrency in task execution, and therefore application logic can be reconfigured to improve performance.), etc.

- *User*, the preferences, the tasks, and the associated goals. The knowledge of user preferences, tasks and goals can enable the activation of an opportune migration as soon as some conditions occur in the current context.
- *Physical Environment*, example attributes are noise, light.
- *Social Environment*, relations among the users that can affect the interaction.

The information may be measured directly, e.g. information about the physical environment like light or noise may be measured by sensors locally or remote. Other types of relevant information, e.g. with respect to user activity may need to be inferred or derived from existing context information, as such information may not necessarily be directly measurable. Thus, a context management framework for OPEN would need to support not only the collection and distribution of context information, but also online processing capabilities for inferred context information in order to ensure the best triggering mechanisms for migration.

5. ARCHITECTURAL FRAMEWORK

This section introduces the architectural framework of the OPEN migration platform, which will be detailed in the following sections. As discussed in Section 3 , the OPEN platform enables three main capabilities of migratory applications: migration of the user interface, or components thereof, application logic reconfiguration and network reconfiguration. As discussed previously these capabilities enable migration, where *migration* is defined as *device change + adaptation + continuity*. The platform supports the migration phases detailed in Section 3.2 and utilizes a context management framework (CMF) as a knowledge base for making decisions or suggestions about how, when and where to migrate.

The OPEN architecture has been derived through a mixture of top-down analysis starting from the application scenarios and requirements, and bottom up analysis starting with the identification of relevant functions based on the experience of the partners.

It is a service-oriented architecture. The first version will be based on a client-server model, but the possibility of a distributed peer-to-peer (P2P) architecture exists as explained later.

Basically, our architecture foresees a number of migratory applications associated with a number of devices, with which the user can interact with the application. An application consists of components, which can be of two basic types: interactive (user interface) and logical (computational).

From a high-level view, as illustrated in Figure 3, each application acts as a client of the OPEN platform, which acts as a server. The Open Client may actually encompass an application on one device and an application server on another, as indicated by the Open Client in the middle of Figure 3. But, it is not always possible to modify servers to make them OPEN aware, thus the architecture enables a developer to turn an application into a migratory application without modifying application servers. The OPEN architecture is also designed to work with a wide range of application platforms, e.g., Web applications, Microsoft Silverlight applications, Java applications and multicore applications. It is also designed to minimize the effort required to turn an application into a migratory application, which it does by providing developers with a set of ready-to-use functionalities.

By using the Open client interface, applications can request migrations. The Open server, in turn, completes migration requests by performing adaptations, and by instructing the target clients to execute the adapted migrating application components after restoring their states. In this way, the user has task continuity after the migration. Thus, with the help of the platform, migratory applications are able to adapt to changes in context, e.g., different devices, different networks or the presence of other users.

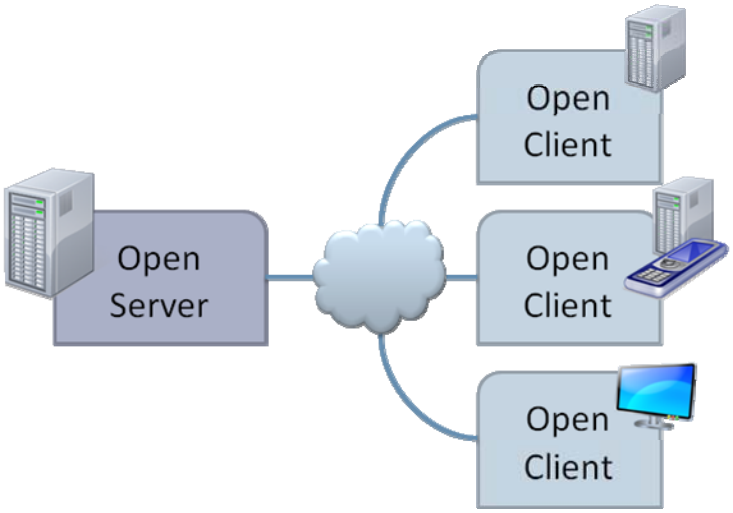

Figure 3 Overall architecture, where applications are seen as clients of the migration platform

The Open server may reside on any device, so long as it is reachable by the Open Clients and fulfills the necessary hardware specifications. Within the scope of the project, a single Open Server is considered where all Clients are registered. It is, however, a small step towards more distributed deployments, such as the one shown in Figure 4. In this example, an organization in an office building (domain 1) runs an Open Server, while a home user has set up an Open Server at her home (domain 2). Open Clients (marked OC) are registered to a particular Open Server, but might handover as the user moves from one domain to the other. The finer points of these mechanisms are not covered by this project, but numerous solutions exist, such as providing Open Server addresses as a field in the DHCP messages when entering the network.

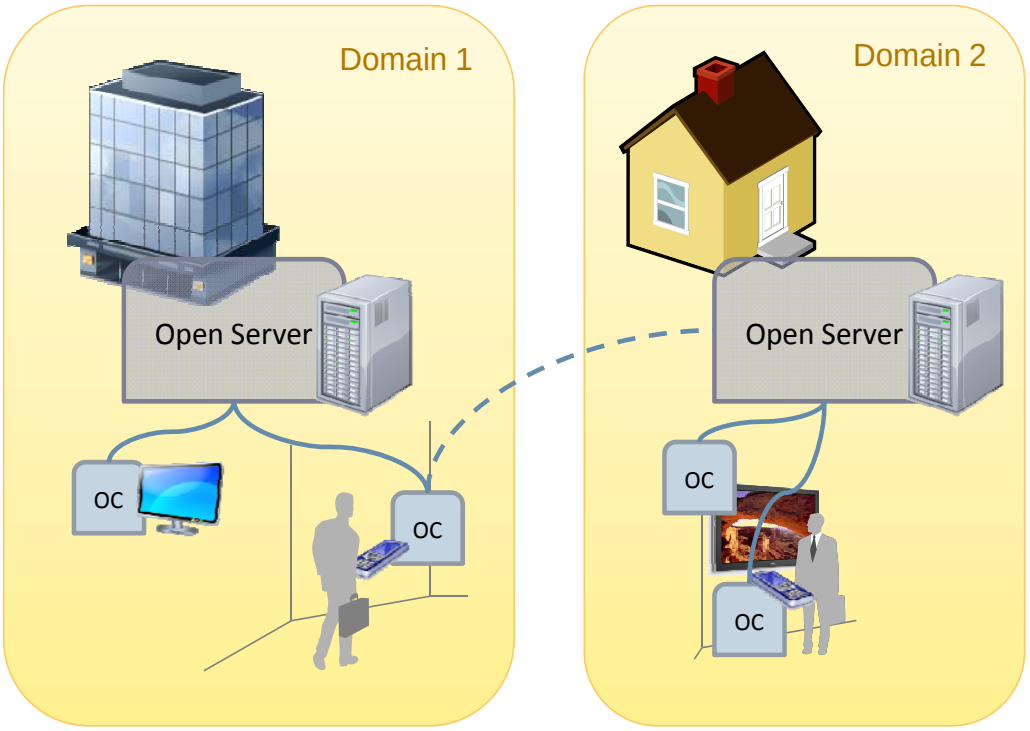

Figure 4 Open Servers may be localized, and clients might handover between them

Likewise, a P2P approach would be possible in an environment where a node is elected among its peers to play the role of the Open Server provided it has the necessary hardware requirements. Once the overlay network is established, the Open mechanisms would continue to work in the fashion described in this document.

5.1. FUNCTIONAL COMPONENTS

Figure 5 depicts some of the major components of the architecture. The platform components are divided into two types: *Adaptation Layer* and *Supporting.* The difference between them is mainly that the first type is more closely tied to the type of application, and, therefore, may be application specific, e.g. specific to Web applications.

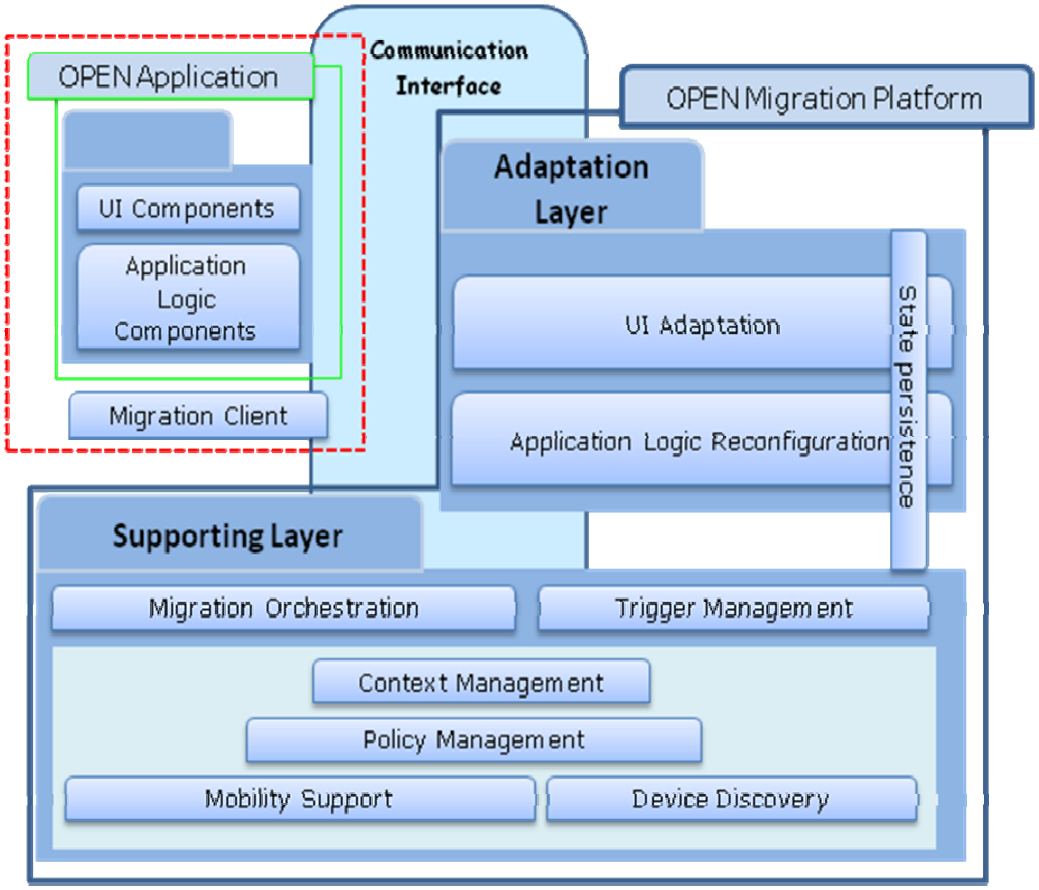

Figure 5 Overview of Architecture and Major Components

The *Migration Client* is a client-side component that can discover the devices in the vicinity. Through it the user can see the available device(s), select a target device(s), and trigger a migration. This is done by sending the migration request to *Orchestration Management* to perform the UI migration process, described in Sections 3.2 and 4. *Orchestration Management*, in turn, utilizes the components *UI Adaptation* and *State Persistence*. The first is responsible for UI adaptation, the last for preserving the application state so that task continuity is maintained.

If there has been a change in the underlying networks, the *Mobility* component is used to adapt to it. In essence, this component performs any necessary network reconfiguration. This may be done as part of UI migration, or independently as part of adapting to changes in network connectivity or quality.

Figure 6 shows some of the major interactions between components, following a manual user request for migration. First, the *application* (user) triggers *Migration Orchestration* to start the migration to a specific target device. *Migration Orchestration*, then coordinates the souce and target *applications*, *UI Adaptation*, *State Persistence*, *Mobility* and if necessary, *Application Logic Reconfiguration* (*ALR)* to achieve the migration.

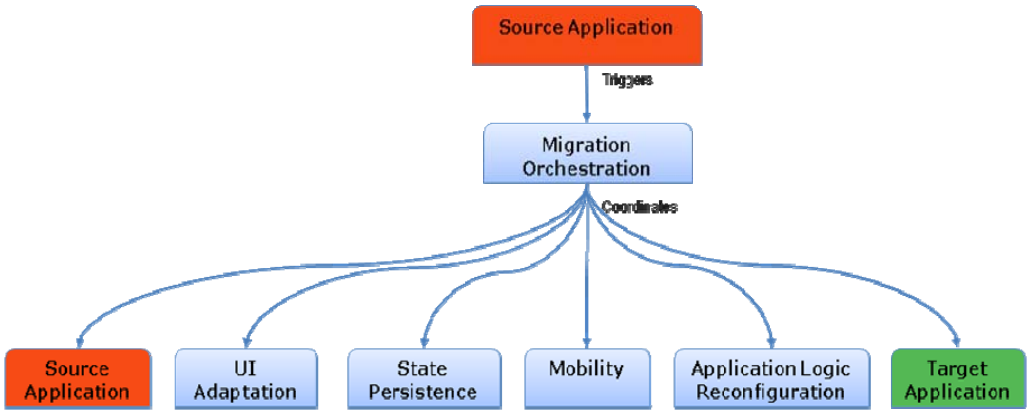

Figure 6 Interactions of components following a user-initiated trigger

Triggers can also be initiated by the system; these are called automated triggers. Figure 7 shows the major interactions for such an automated trigger. *Device Discovery* and other sources of context information interact with *Context Management* to update the context. As context changes, *Context Management* notifies *Trigger Management* of any changes in context, for which Trigger Management had previously registered its interest. If this change of context triggers a trigger-rule, the resulting migration is checked against policies with *Policy Management*, and is only initiated if it conforms to policies. After that the interactions run the same course as for a manually triggered migration. More detailed diagrams of component interactions are given in Section 9.

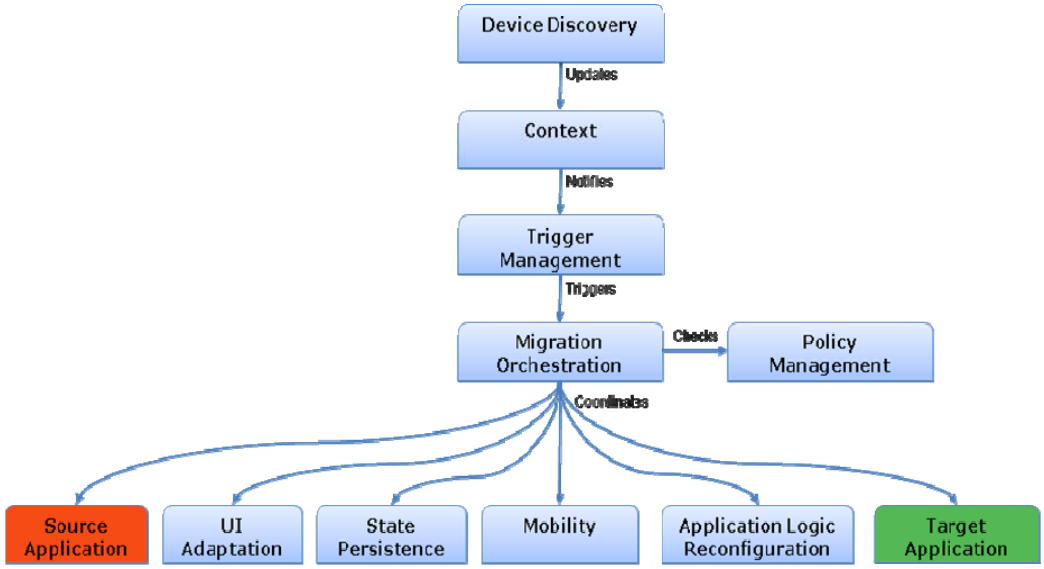

Figure 7 Interactions of components following an automatic (system-initiated) trigger

T

In the next sections we provide more detailed descriptions of the three main parts composing the OPEN Platform according to the three main aspects that are handled: user interface migration, application logic reconfiguration, and supporting functions like *Mobility*. Section 8 covers all the supporting functions. This organization is represented graphically in Figure 8.

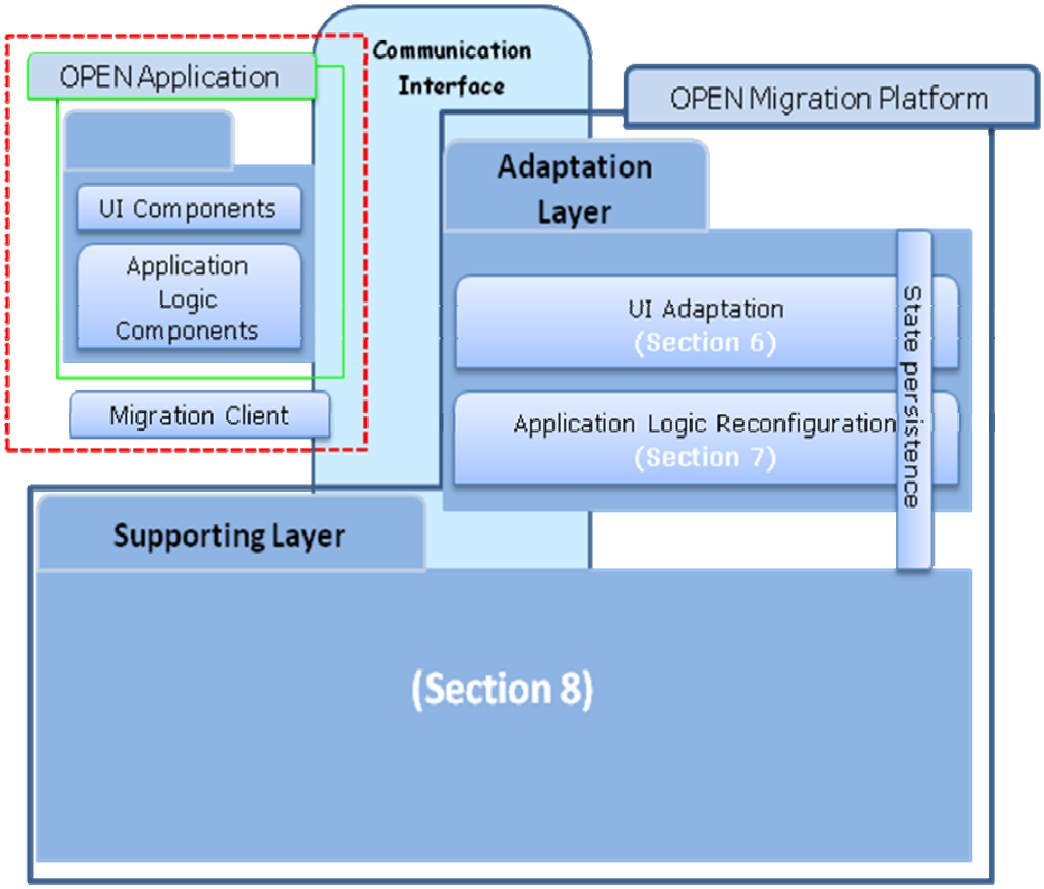

Figure 8 How Document Sections relate to Components

6. USER INTERFACE MIGRATION

The main components of user interface migration are user interface adaptation (*UI Adaptation*) and the preservation of application state (*State Persistence*) when switching devices. These are discussed in general, and then specifically, first for Web applications, and then for multicore applications.

Some guiding principles of our work in this area are:

- *Avoid manual solutions*, in which developers directly create migratory versions of an application. Although manual solutions give full control over the results, they lack generality and would be too expensive when applied to various applications;
- *Leverage existing Web Applications*, while we would like to make all existing applications migratory, this can be too ambitious for this project. Thus we decided to focus on existing web desktop applications. These are applications that can be accessed through a Web browser. The reasons for this choice are that this type of application is the most common (there exist millions of Web desktop applications). In addition, we can manipulate and transform the content of such user interfaces to obtain versions suitable for various types of devices;
- *Target a wide variety of interaction platforms*, we want to be able to dynamically create migratory user interfaces adapted for various types of devices and implementation languages (even non Web languages, which are languages that allow specifications that can be interpreted by Web browsers) starting with existing desktop Web applications.

6.1. ADAPTATION STRATEGIES

The version of an application, which has been adapted for the target device of a migration, can be obtained and managed in various ways:

Pre-computed, in this case the application version has been defined in an ad-hoc manner, prior to the migration process. Usually this implies more control over the resulting version. However, this solution requires considerable development effort (the development of a version for each type of target device for each application considered, consequently it has a limited applicability. Such pre-computed versions can be either preinstalled in the target device or loaded on-demand when migration occurs.

Dynamically generated, in this case the application version for the target device does not exist beforehand; it is automatically generated when migration occurs. The new version is obtained by applying some transformation rules coded in the adaptation engine starting with the information of the application version in the source device. This solution requires limited effort (it only requires the activation of the automatic adaptation engine). However, there is less control on the resulting version and it is difficult to find a set of rules that provide optimal solutions in a wide set of cases. One example of a strategy implementing this approach is based on three main phases: reverse engineering to create a logical description of the interactive application from the existing implementation, semantic redesign to adapt the logical description to the target platform, adapted implementation generation. This is the solution for adapting the user interface of a

migrating web application which is currently followed in the project, and which has been implemented in a prototype implemented in Java under further development. In this prototype, the design of the considered application is represented by a XML-based logical description, which is dynamically adapted according to the characteristics of the target device, and then it is used to generate the final user interface for the considered platform. Further details of this approach are explained in [D2.2].

Mix between pre-computed and automated, in this case various levels of automation are considered, but starting with some aspects pre-computed. This ranges from developing the source application according to some guidelines that make it more suitable for adaptation to having the structure of the adapted version pre-defined with some aspects that are dynamically filled in it. In some cases there is one generic version of the application with annotations for obtaining adapted versions to different types of devices.

In general, there is no a priori 'best' solution to be selected among such strategies, although on the one hand a pre-computed solution will emphasize the full control by the designers in obtaining the adapted version, but it requires a big effort for addressing various applications and target devices because each application version must be manually created. On the other hand, a completely dynamically generated solution requires less effort but it may generate non optimal solutions in terms of usability, because the underlying rules driving the generation may not be sufficiently able to address the specific usability requirements of each application.

In line with the principles of the project, listed at the beginning of Section 6, we decided to invest most of our effort into dynamic automatic generation applied to Web applications. This aligns well with our principles of *avoiding manual solutions* and *leveraging existing Web applications*. The UI Adaptation functionality described in Section 6.3 is based on this strategy. At the same time the project is continuing to consider the possibility to modify the rules driving adaptation and to provide guidelines for designing and implementing an interactive application in order to make it more easily manageable by our platform.

6.2. WHERE ADAPTATION CAN TAKE PLACE

Adaptation can take place in at least three different types of hosts, which are listed below. Our final architecture uses the Intermediate server option to avoid the two main drawbacks of the other options, installation on all application servers and performance limitations of clients.

Types of Host

The Application Server, which recognizes the type of target device and activates an adaptation accordingly. The drawback of this solution is that the UI Adaptation component must be duplicated in all the potential application servers.

Intermediate server, e.g., the OPEN Migration Server, in this case UI Adaptation is located in a single server that acts also as a proxy server monitoring the requests from the target client device and adapting the results according its resources. Potential issues for this architectural approach are congestion or lack of scalability. An example of this approach will be better explained in Section 9.2, where we describe the various steps of the approach.

Client device, in this option the UI Adaptation component is installed in the target client device and is applied when migration has been triggered. The local installation guarantees that there is

full access to information on the local device capabilities. However, with some devices with limited capabilities (such as some types of mobile devices) there can be problems in supporting the processing requested by adaptation.

Every architectural approach also impacts the business ecosystem that needs to be put in place:

- If the adaptation takes place at application server level, the focus will be on the specification and promotion to the developer community of a framework complete with guidelines and development constraints.
- If the adaptation is based on an intermediate server, the effort related to the complexity and computational resources for the user interface adaptation should be allocated to an ISP or telco operator who should provide the intermediate server.
- If the adaptation takes place in the client device, this could impose some form of device manufacturers' endorsement, in order to ensure that the UI Adaptation component can best exploit the capabilities offered by the device.

In addition, it is worth pointing out that the aspects that are related to *where* adaptation should occur have an impact on issues related to adaptation strategies (*how* to adapt) described in the previous section. Indeed, for instance, a heavy adaptation strategy like the dynamically generated one is difficult to combine with an adaptation that takes place in the client device, when the client platform is a thin device with limited performance capabilities.

6.3. USER INTERFACE ADAPTATION AND CONTINUITY OF WEB APPLICATIONS

There are four components, shown in Figure 9, that support the dynamic generation strategy for adapting migratory Web applications. Adaptation is based on logical descriptions of user interfaces, specified using XML-based languages. In such descriptions there is an abstract level, which is platform-independent and a concrete level, which refines the previous one by adding platform-dependent elements and attributes. The four components which support adaptation and task continuity are:

- *Reverse Engineering*, this takes the existing Web pages for desktop systems and builds the corresponding logical descriptions;
- *Semantic Redesign*, this module is in charge of performing the adaptation to the target device. For this purpose it takes the abstract elements identified by the reverse engineering module and maps them into concrete elements more suitable for the target device. It also splits the source presentations into multiple presentations, if they are too expensive for the interaction resources of such target device.
- *State Mapper*, (a realization of *State Persistence* for Web applications) once a concrete description for the target device has been obtained, the state resulting from the user interactions in the source user interface is associated with it. The abstract elements are used to identify which concrete elements in the source interface correspond to the concrete elements in the target interface. *State Mapper* has a client-side counterpart, which collects the state information. Section 6.4 gives details about the state information which can be collected and mapped.

- *User Interface Generator*, this module generates the user interface in some implementation language. One concrete description for a given platform, for example, a graphical form-based interface, can be associated with various implementation languages (such as Java, XHTML, C#). The generated user interface is then uploaded on the target device.

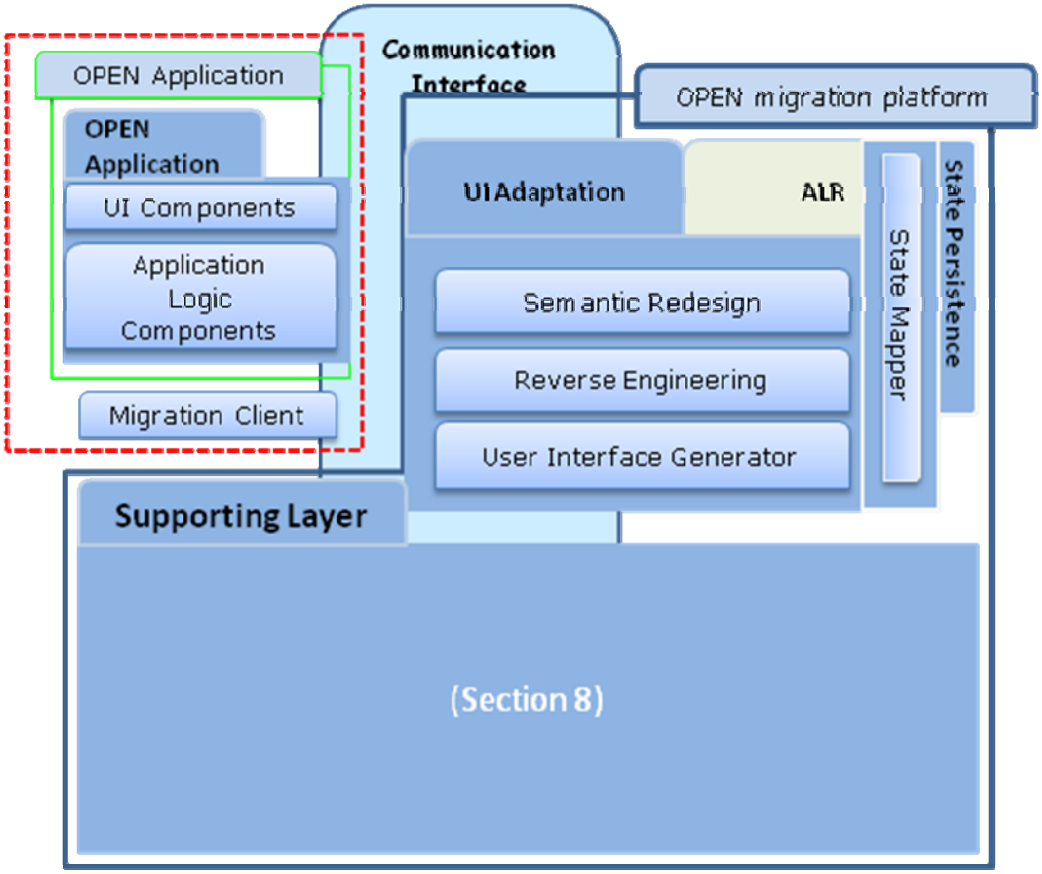

Figure 9 Components for UI Adaptation and Continuity of Web applications

Figure 10 is a representation of how *Semantic Redesign* and *State Mapper* use the abstract and concrete descriptions of the source and target UIs in order to adapt the source UI to the target device, and to guarantee that the user experiences task continuity.

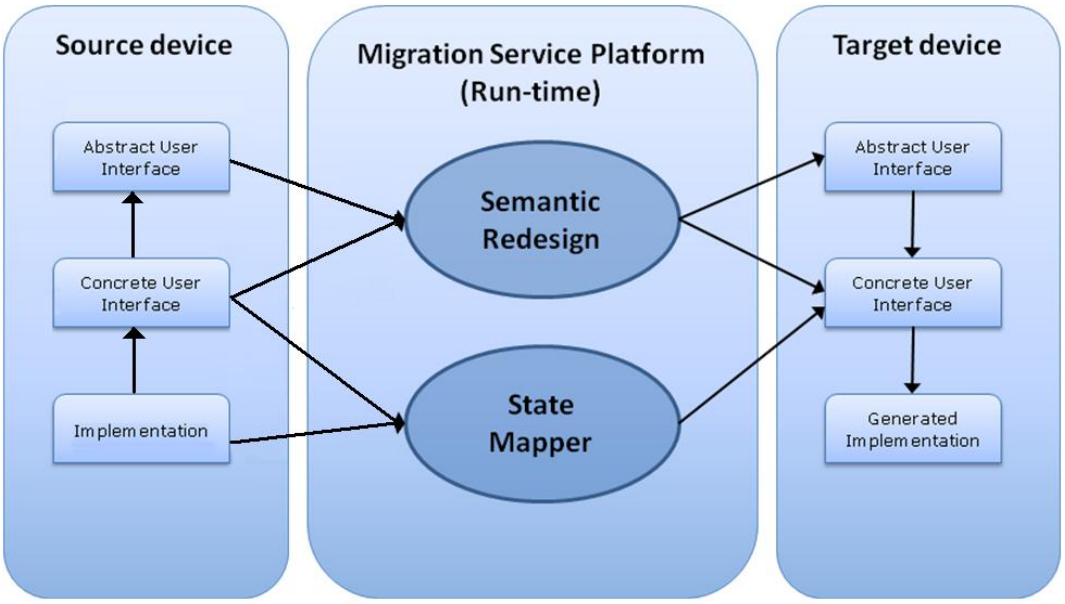

Figure 10 UI Adaptation Process for Web applications: Adaptation and Continuity

6.4. CONTINUITY SUPPORT FOR WEB APPLICATIONS

One of the main components of the migration platform is the support for task continuity, *State Persistence*. A realization of *State Persistence*, created especially for Web applications is called *State Mapper.* Its goal is to allow users to change device and to continue the task at hand from the point they left off in the source device. This implies the ability to persist the state of the source application and to apply it to the version that is going to be activated in the target device.

The state of the interactive application can be seen as logically composed of the state of the user interface (which is defined by the information entered or selected by the user) and the state of the application logic.

In a Web application, we have identified at least eight aspects that can be relevant for defining the state of Web user interfaces, and which can have an impact on the overall user experience. The first element is associated with the user input. People make selections, enter text and modify the state of various input controls during a session, and such modifications should not be lost when moving to a new device if we want to maintain continuity. An associated element is the set of client-side variables associated with small functionalities (e.g. Javascript variables).

Another component that can be dynamically modified is the content of a Web application. While this can be easily managed with dynamic Web sites using PHP, JSP and similar languages because whenever a new request is performed, a different page is uploaded, with Ajax scripts this aspect becomes more problematic. Indeed, in this case the content of the page can vary without requiring the loading of a new page. Thus, it becomes more complex to detect what is actually composing the currently displayed page.

Cookies are more and more used, and they allow an application to provide small pieces of information to the client in such a way that whenever the client accesses the application, then the client identifiers are inserted in the HTTP protocol. It is important that, if and when a user changes device, then the current application preserves the same cookies in order to be recognized by the application server. A related technique is the session: it is a server-side mechanism, which stores information related to the user session, which is, in turn, associated with a specific identifier.

Another important aspect is the history of user accesses, which is maintained by the Web browser and drives the behavior of the frequently used back button in the browser. Since the user is still the same, even if she has changed device, then she would appreciate still being able to easily return to recently accessed pages, even if through a different device. It is clear that the pages accessed through the new device may be adapted to the currently available interaction resources. In some cases (e.g. migration from desktop to mobile), it may even happen that the original desktop page is split into multiple mobile pages, thus accessing all its content may require further navigation.

Bookmarks are another interesting aspect that can be considered part of the user interface state. Users often use them to quickly find and access favorite pages. In migration, the devices change but not the user, who still has the same interests and may appreciate the possibility to use current bookmarks, including the pages that were bookmarked in the previous device. Another element that has similar characteristics is the browser home page: in some cases users may be interested to migrate it to as well, regardless of platform.

A last element that can be considered part of the state is the query string included in a URL after the "?" symbol. It is usually used to specify parameters for a dynamic site, which define some data that are presented in the associated page. By modifying the query string we will access the same Web site, but since the parameters vary, the corresponding page varies in terms of content.

In D2.2 we discuss in more detail how the OPEN platform addresses the rich user interface information state discussed in this section.

6.5. USER INTERFACE ADAPTATION AND CONTINUITY OF MULTICORE APPLICATIONS

The previous section considered migration of Web applications. In this section we consider another type of application. In particular, we describe a specific framework for application implementation that is being developed within the OPEN project, the Multicore UI Toolkit. This toolkit can be used together with the OPEN MSP to implement migratory applications which have high-end, multimedia graphical interfaces.

6.5.1. THE MULTCORE TOOLKIT

The Multicore UI Toolkit is a library of classes that enable application programmers to implement visually appealing graphical user interfaces similar in look and feel to existing UIs on mobile devices (e.g. Apple iPhone), game consoles (e.g. Xbox Live! Arcade UI, Playstation™ Store) or set-top boxes (e.g. Channel list, program guides, etc.).

The toolkit is mainly targeted at multimedia applications and in particular will provide a 3D compositing layer allowing for features such as rotated and scaled windows, transparency and animated widgets.

To achieve these effects and to provide the best possible performance, the library uses multiple threads internally to leverage multicore CPUs.

The innovative adaptive load-balancing implementation always utilizes the full processing performance provided by the CPU.

In particular, the system automatically adapts to the changing number of available cores at runtime, which is a key advantage with respect to application migration.

It also supports disabling of CPU cores at runtime, without interruption of the running application and with no special requirements on the programmer's side. This behavior is especially useful for mobile and embedded devices, as application performance can be adapted with respect to power consumption and battery life. This is the most important distinguishing point of the toolkit as none of the existing GUI libraries on the market are designed from the ground up to support multicore rendering on CPUs to this extent (Apple and Microsoft focus on GPU acceleration for example).

The second important aspect is ease of adaptation and migration of GUI layouts to different devices: For example, all relevant coordinates, regions and parameters are specified in normalized floating point coordinates which enables GUIs to be resolution independent. Furthermore, memory footprint and run-time resource requirements are very low in order to ensure smooth interaction on mobile and embedded devices.

The GUI toolkit is a software development kit similar to e.g. Trolltech Qt / Qtopia (see http://trolltech.com/) or wxWidgets.

Viewed from a higher level it provides the following functions:

- Widget implementations (for windows, buttons, lists, etc.)
- handling state and updates, messaging logic and event handling
    - E.g. when the user moves the mouse, clicks a button, scrolls a window, etc. the corresponding event is stored in a message queue
    - On update, widgets get their events from the queue and react to these events, i.e. change their state
- rendering and compositing
- transformation (rotation, scaling)

A Renderer component could provide lower-level graphics tasks such as drawing of graphical primitives like diagrams, charts, etc.

The Compositor sub-system performs several functions, including

- control of display update
- management of the active windows and update of their transformations
- interaction with the rendering and multicore sub-systems to draw the final layout

The following figure (Figure 11) gives a brief high-level overview of how various components interact in the generation of the user interface for a multicore platform. The output of the *Adaptation Tool* is an XML *concrete user interface description*, which is used as a starting point for the generation of a user interface exploiting the multicore library.

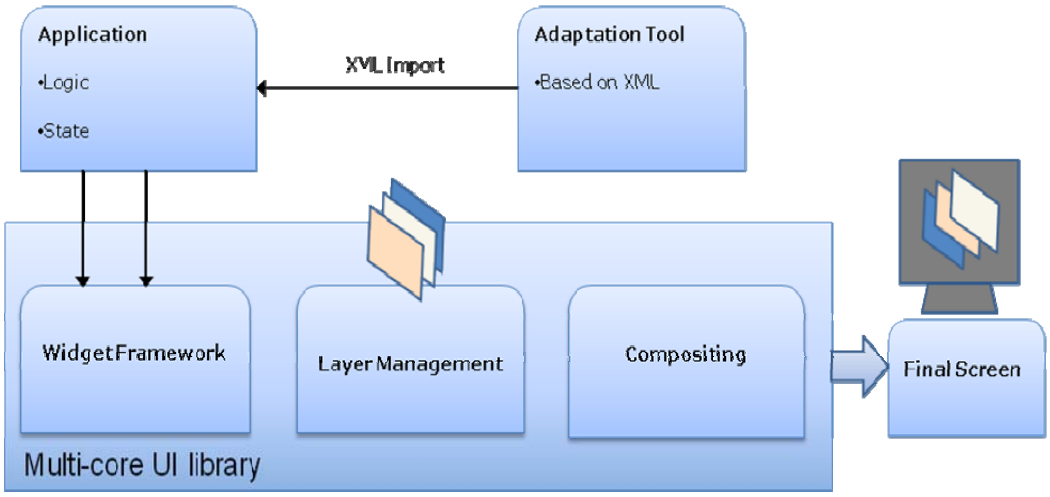

Figure 11 Multicore Applications: Overview of how various components interact

Besides instantiating new GUI widgets and handling events, applications will also to some extent have to interface with the toolkit's Renderer and Compositor sub-systems in order to control display update and possibly perform lower-level graphics tasks.

6.5.2. ADAPTATION AND CONTINUITY OF MULTICORE APPLICATIONS

Adaptation of multicore applications is analogous to adaptation of Web applications, except that there is no platform-independent abstract level representation of the user interface. There is only a concrete level description, the XML *concrete user interface description*.

An multicore application creates its GUI elements and layout using the Multicore Toolkit classes which correspond to the XML *concrete user interface description* provided by the *UI Adaptation Tool*. As shown in Figure 12, this tool is a type of *UI Adaptation* component, which is specific to multicore applications.

State persistence for multicore applications is complex. Although some tasks regarding management of state and logic are already provided by the GUI framework, typically large parts

of these tasks also have to be implemented on the application side. This results in tight intertwining of GUI and application logic code. However, the GUI library will not perform migration of data and state information stored inside GUI widgets; it is the responsibility of an entity external to the GUI library, the *client-side counterpart of State Persistence*, to get these values from the respective widgets and hand them to the OPEN platform. Figure 12 gives an overview of the components of Multicore UI Adaptation and State Persistence, the later is realized by a component specific to multicore applications called the *Multicore State Mapper*.

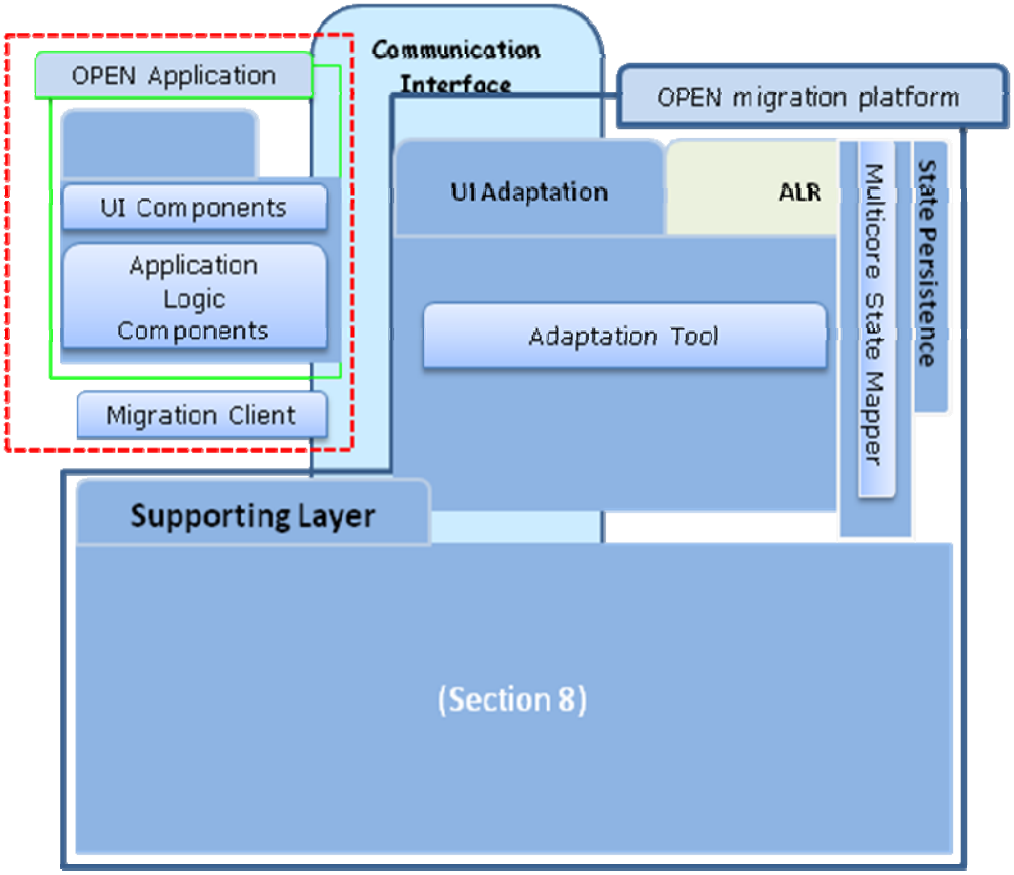

Figure 12 Components for for UI Adaptation and Continuity of multicore applications

The process of adaptation for multicore applications, shown in Figure 11, is analogous to the process for Web applications, shown in Figure 10. Figure 11 is a representation of this process, showing that the *Adaptation Tool* and the *Multicore State Mapper* use the concrete descriptions of the source and target UIs in order to adapt the source UI to the target device, and to guarantee that the user experiences task continuity.

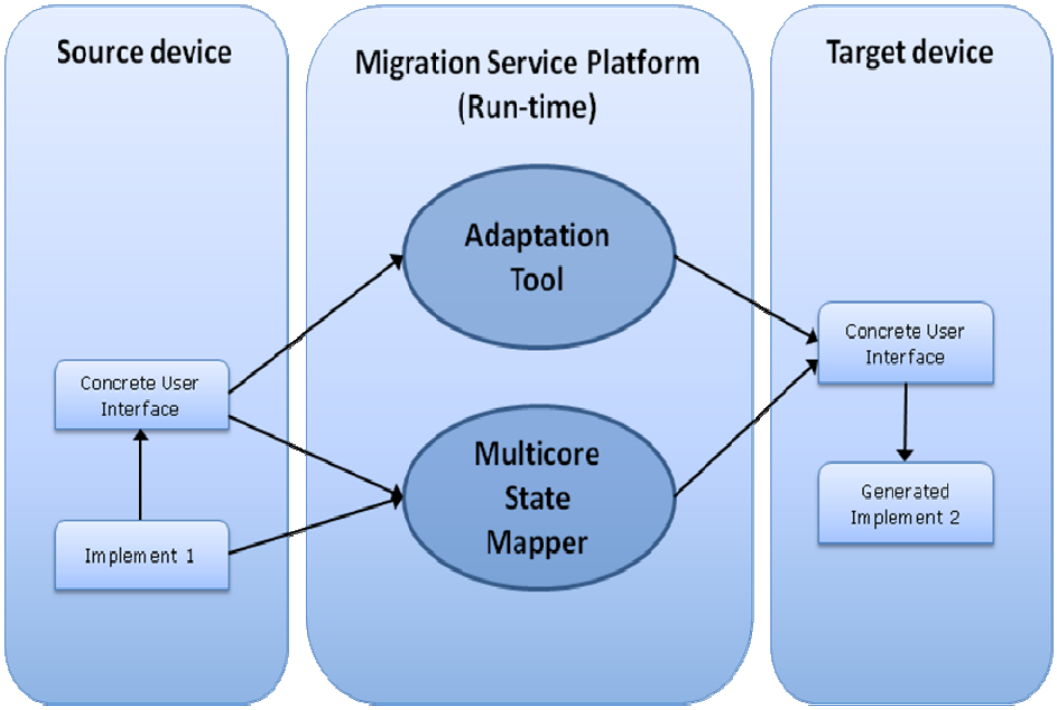

Figure 13 UI Adaptation Process for multicore applications: Adaptation and Continuity

7. SUPPORTING APPLICATION LOGIC RECONFIGURATION

Application Logic Reconfiguration (ALR) is about adapting the behavior of the application at runtime according to changing context information or user preferences. In the first section we will introduce which types of ALR we will consider, and how they interact with other components. The *ALR components* and some related components are shown in Figure 14. Afterward discussing the types of ALR, we will give a rough overview of the interplay between ALR and migration.

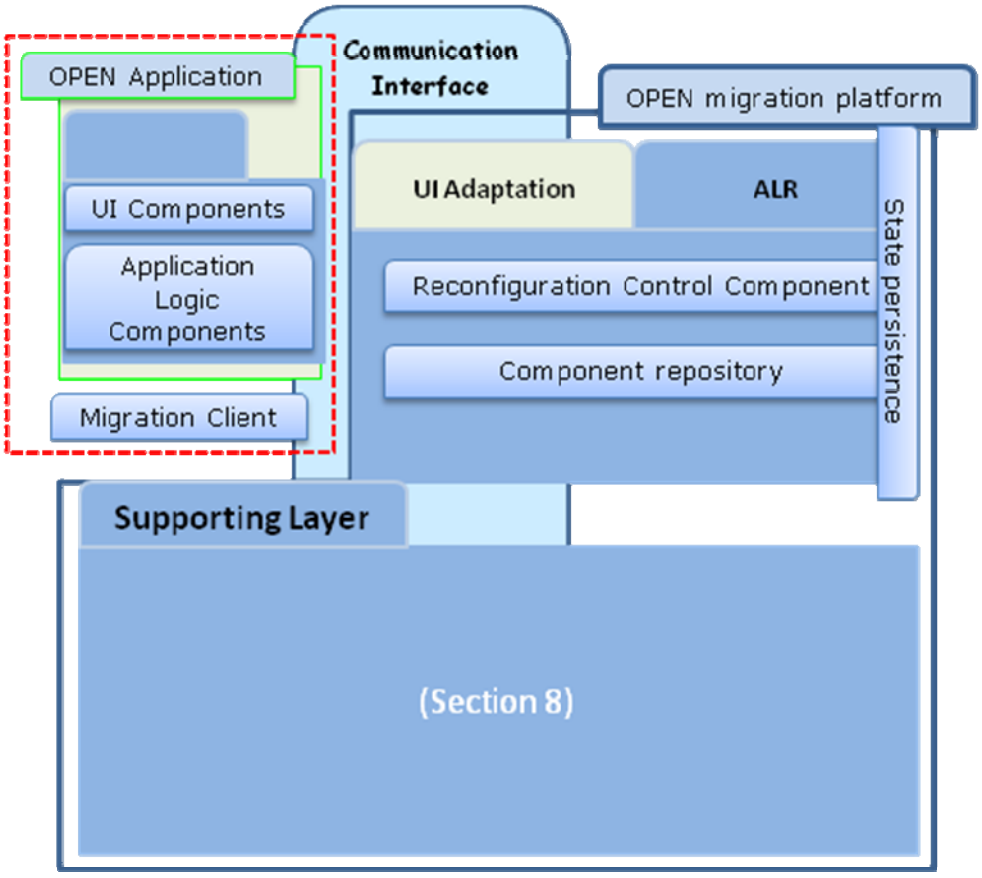

Figure 14 Major Components related to ALR

Figure 15 represents an example of a combined migration and logic reconfiguration, in which a PacMan game is migrated from a PC to a PDA. To adjust to the limited screen size of the PDA, the UI is adapted so that fewer dots are displayed.

In addition, because steering the PacMan is more difficult on the PDA, either the speed/intelligence of the ghosts is reduced or the scoring is adapted, e.g., higher scores per dot are rewarded. These modifications are achieved by means of application logic reconfiguration.

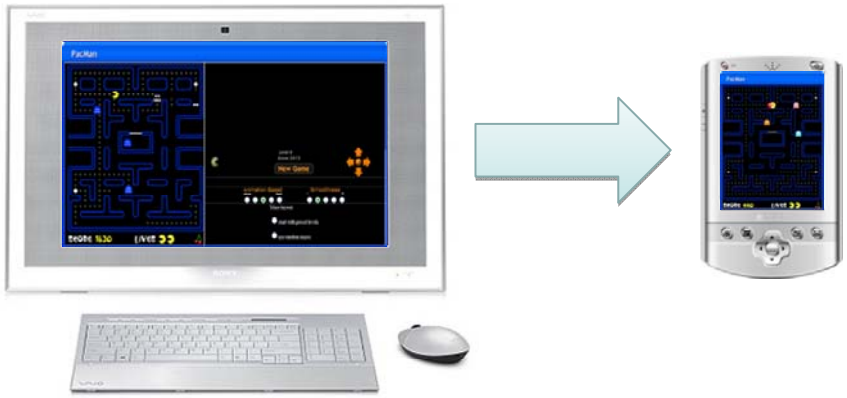

Figure 15 Example of Migration and Application Logic Reconfiguration

7.1. APPLICATION LOGIC RECONFIGURATION OF SERVICE-BASED DYNAMIC ADAPTIVE SYSTEMS

We consider dynamic-adaptive systems as systems built from a set of services that work together to perform tasks that are useful for application users. In addition, compared to established component-based applications, the behavior of dynamic-adaptive applications adapts during runtime to the needs of the current user and his environment.

Components are software entities that realize specific services that are described by and accessed through interfaces. Other components can define dependencies to provided services by defining them as required. Provided and required services have to be linked together in order to enable, for example, method calls from one component to another. Figure 16 shows a set of component instances running on a PC that offer and require different services, described by interfaces. Required services are depicted as semi-circles, while provided services are depicted as circles. That notion follows the UML 2 standard [UML]. In order to run the application, corresponding provided and required services have to be connected. The decision about which services to connect will be made by the *Reconfiguration Control* Component shown in Figure 14.

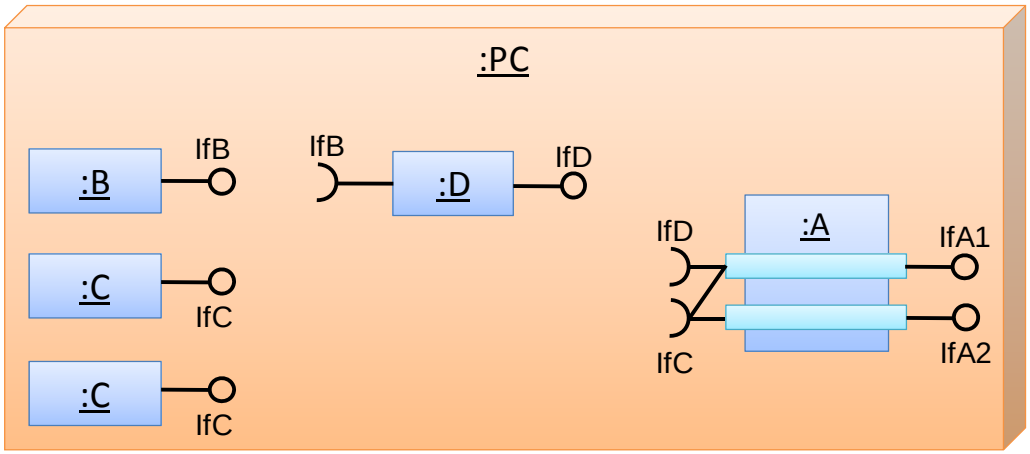

Figure 16 Component instances running on a PC ready for wiring and interacting together. If a component instance holds all required references to service implementations, it becomes executable and will provide the given services to other components.

Therefore, the task of the *ALR* is to ensure that the wiring of component instances always fits to the user's needs on the one hand, and to the usage context on the other hand. The wiring can be changed during runtime based on context information, user preferences, or currently available component instances. The rewiring of components is one aspect of the considered reconfiguration types.

Currently we distinguish three types of reconfiguration [NKA+07]. The first one we call *Service Implementation Adaptation*, where the behavior of a single service changes by replacement of the implementation during runtime based on any available context information. One example is that a speech recognition service adapts its recognition algorithm based on the emotional state of the user. This could be useful because the speech of a relaxed person is very different than that of a stressed person and therefore there would be a lack of reliability in changing situations. This type of adaptation could be triggered by the *ALR* and performed by the application component itself.

The second type of adaptation we call *Service Usage Adaptation* which is depicted in Figure 17.

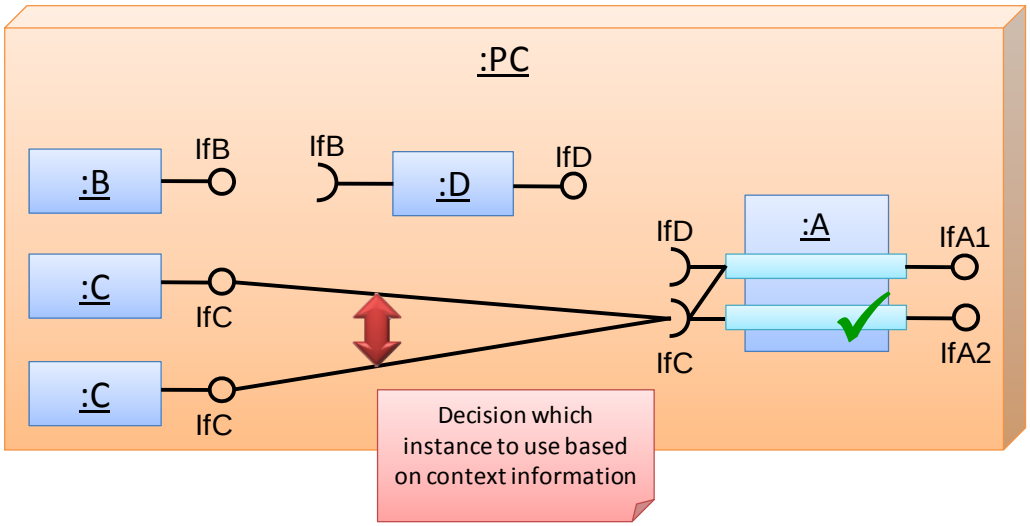

Figure 17 This figure illustrates the Service Usage Adaptation. If two implementations of the same service are available, the middleware has to decide which one to use in the current situation.

In this type of adaptation an application component instance, which is currently in use is replaced by another application component instance, which is more appropriate in the current context. A very powerful but resource-intensive component instance could, for example, be exchanged for a less comprehensive but more energy-saving component instance, if the battery of the device is getting low. This type of adaptation is performed by the ALR.

The third type of reconfiguration we call *Component Configuration Adaptation* which is illustrated in the next figure (Figure 18).

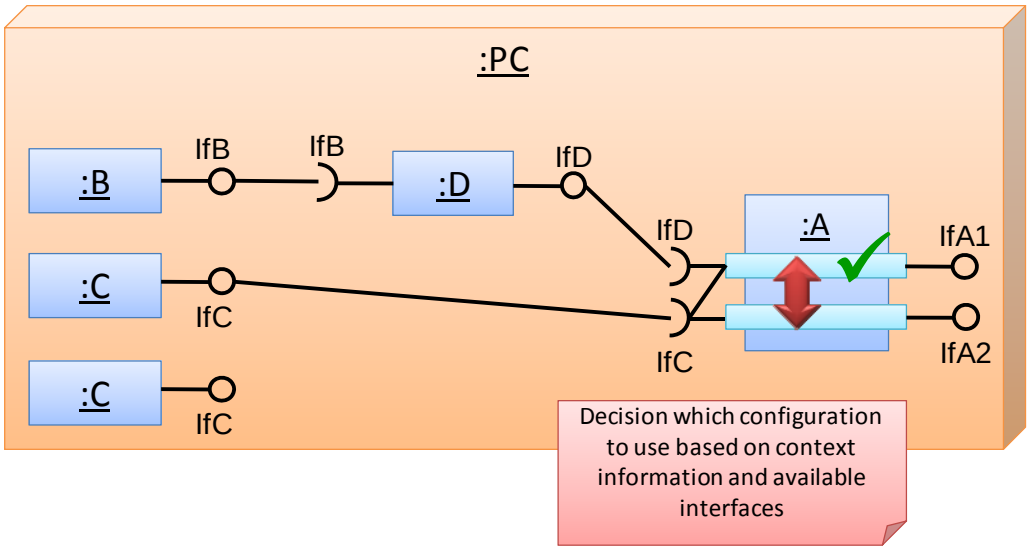

Figure 18 The figure illustrates the component configuration adaptation. If interfaces IfC and IfD are available, component A provides IfA1 to other components, if only IfC is available, component A will provide IfA2. The active

configuration is tagged with a green check mark. Depending on which application component configuration is active, the behavior of that component may be different.

In this case, a component can consist of several configurations as shown in component A in Figure 18. Each configuration is a mapping between provided and required services. If all required services of a configuration are available, this configuration will get activated and the provided services of that configuration will be available to other components in the system. In addition, the behavior may change depending on which configuration is currently active. The active configuration is tagged with a green check mark. That means that during reconfiguration the functionality of the application can change due to the instantiation of new components with new services.

The *ALR* Component realizes these types of adaptation. The main trigger for adaptation will be the migration of application components, and the change of context information. To do this, the *ALR* Component interacts with several components from the OPEN architectural framework shown in Figure 14. Next, we will shortly describe how the *ALR* cooperates with these.

- **Access to the Component Repository**

  Each component instance within an OPEN system has to be published at the *Component Repository*. In order to decide which of the three types of adaptation can be used, *Reconfiguration Control* has to access the component repository in order to decide which instance is the best in the current situation. *Reconfiguration Control* is also interested in the events like new component instances, and therefore new services, become available or other component instances disappear. Therefore, it will register for these kinds of events at the *Component Repository*.

- **Interaction with Trigger Management**

  A trigger is an event which tells the system that a migration has to be performed. Therefore, it will also be used to initiate the adaptation of the application. To do this, specific trigger definitions will be stored at the *Trigger Management*, that is conditions when to trigger migration and adaptation. A trigger could, for example, define that the application or parts of it have to be adapted if the battery power is below 50%, below 30%, and below 10%.

- **Interaction with Migration Orchestration**

  When one or more application components are migrated, an adaptation of the application logic could be required. To do this, when a migration is performed, *Migration Orchestration* interacts with *ALR* to adapt the application components for the target device.

- **Interaction with Context Management**

  The *Context Management* will be used to obtain high-level context information in order to decide how to adapt the application, that is, to decide which type of adaptation to apply and how to realize it. If the battery power, for example, is between 50% and 30%, *Service Implementation Adaptation* could be applied. If the battery power is between 30% and 10%, maybe *Service Usage Adaptation* will be useful.

- **Interaction with Policy Management**

  The *Policy Management* is accessed by application logic reconfiguration in order to obtain specific guidelines or preferences for adaptation. Maybe the user, for example, wishes to slow down the application in case of low battery instead of limiting the functionality.

The first version of application logic reconfiguration deals with services implemented in Java running on Windows or Linux. Since the addressed applications are distributed applications, where services may run on different devices, an appropriate middleware for supporting those kinds of applications was used, OSGi (OPEN SERVICES GATEWAY INITIATIVE). CORBA (Common Object Request Broker Architecture) could also have been used. These existing middleware systems serve as a basis for realizing the communication between all required architectural building blocks used for application logic reconfiguration and application components. Components which originally were not designed to communicate over a specific middleware can be integrated through simple wrappers.

In order to perform reconfiguration, each application component has to implement specific interfaces through which *Reconfiguration Control* can control the lifecycle of the component and initiate reconfiguration tasks. The concrete methods are described in deliverable [D4.1].

Application logic reconfiguration is only one part of adaptation of the application logic. The second part considers the migration of single or all components of an application. In the next section we will therefore explain the role of *Orchestration Management in application logic reconfiguration.*

7.2. INTERPLAY OF APPLICATION MIGRATION AND APPLICATION RECONFIGURATION

The following figure (Figure 19) shows one example of how application migration and application logic reconfiguration could interplay. There is for example a running application on the PC which the user wants to migrate to his PDA. Among other actions, the MSP has to transfer the state and in some cases the code of each component to the PDA and start these components with their states on the target device. These tasks are basically performed by *Orchestration Management*. The application logic reconfiguration will then try to adapt the application to the user's needs and the available resources.

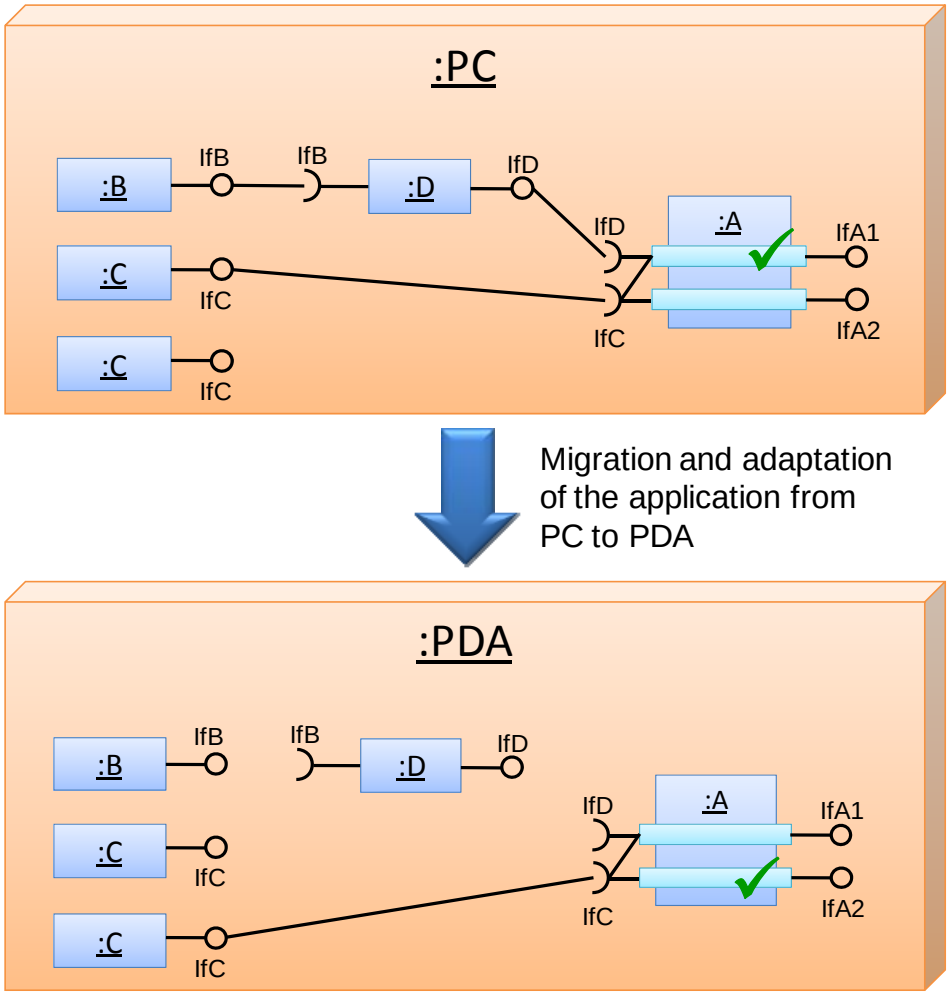

Figure 19 The figure shows the interplay between migration and application logic reconfiguration as described in the section before. In this case *Service Usage Adaptation* and *Component Configuration Adaptation* are performed.

Dependencies upon *Orchestration Management* are the same as described in the previous section. However, the reasons for them are slightly different. *Orchestration Management* is notified by *Trigger Management*, if migration is required. For migration control and management, the location of specific component instances becomes important. Therefore, information about that has to be managed and stored for example at the *Component Repository*. Furthermore, a component, which has to be migrated, has to be stopped, its state has to be stored, its code has to be transferred to the target device, and the component has to be started again.

The communication infrastructure and supported platforms are basically the same as described in the previous section. In addition, the MSP has to initiate and control migration of services and their code. This is done with the aid of the middleware system, OSGi, or CORBA, for instance, which both support mobile code. The communication between application components and MSP components is realized as described in the previous section.

This is only one example of how migration and reconfiguration work together. Many other scenarios are possible. One example is that only parts of the application migrate from one device to another while other components stay at their source device. A more detailed discussion of further migration/adaptation scenarios and middleware solutions is given in deliverables D4.1 (Solutions for application logic reconfiguration) and D4.2 (Migration service platform design).

8. OPEN ARCHITECTURAL FRAMEWORK: SUPPORTING FUNCTIONS

This section completes the description of the architecture by describing all of the supporting functions of the platform. Figure 20 represents the complete architecture, showing the supporting functions, the adaptation layer functions and the migratory application.

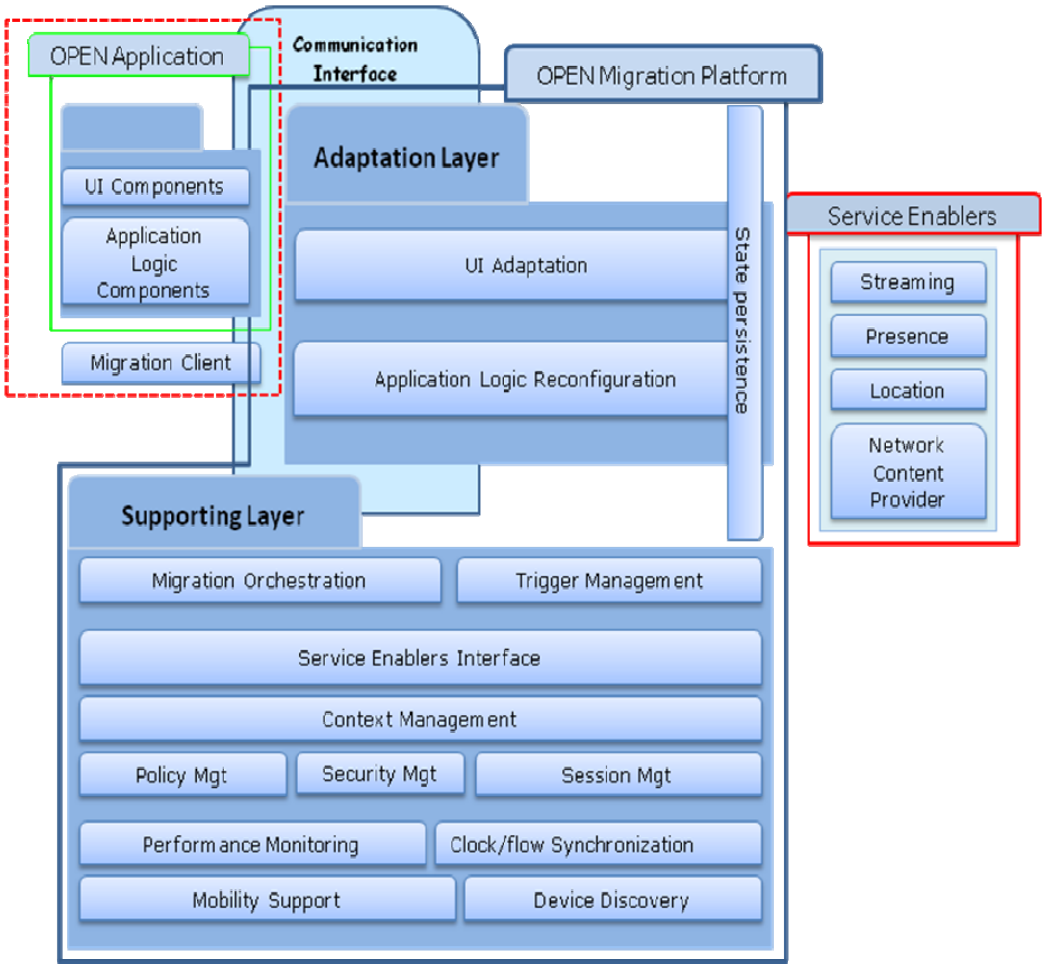

Figure 20 OPEN Architectural Framework

The *supporting functions* of the platform support the adaptation layer functions: UI Adaptaion, State Persistence and ALR. This set of supporting functions was arrived at by careful study of the requirements described in D1.1 and D5.2, and has been discussed and agreed upon by project partners.

The *Service enablers interface* is, as its name suggests, an interface to supporting services that are outside the platform, and which are categorized as service enablers. The *Service Enablers* themselves, since they are external to the platform, are not described in any detail in this document.

In the following sections, an overall description of each of these functions is presented. The description contains: first, a brief functional specification; second, an indication of interactions with other functions; third, if applicable, a brief discussion of potential candidate solutions for the function; and, finally, a list of requirements met by the function.

The descriptions are high-level functional descriptions based on the functional requirements derived in D1.1. In Section 9, some examples are given of how the functions interact on an overall level. The function details (e.g. interfaces, requirements, candidate systems, specific interactions) for the communication and context management middleware are highly dependent on which scenario and network architecture they run in. For this reason, a detailed analysis is given in D3.1 of networking scenarios and consequences of using these migration support functions in the scenarios. Details of the interfaces between components are to be found in D4.2.

8.1. TRIGGER MANAGEMENT

This component analyses contextual information changes and decides whether or not a migration should be activated through issuing triggers. Simple triggers can be provided by the context management function itself. Here, simple thresholds on context information values can be used or even fusion of different pieces of context information may apply.

Even more complex triggers, e.g. based on inferred knowledge from available context information, network information or based on user-input, may be generated and handled by this function. Figure 21 summarizes the interactions between Trigger Management and the major components with which it interacts. It also shows some of the important context information managed by Context Management.

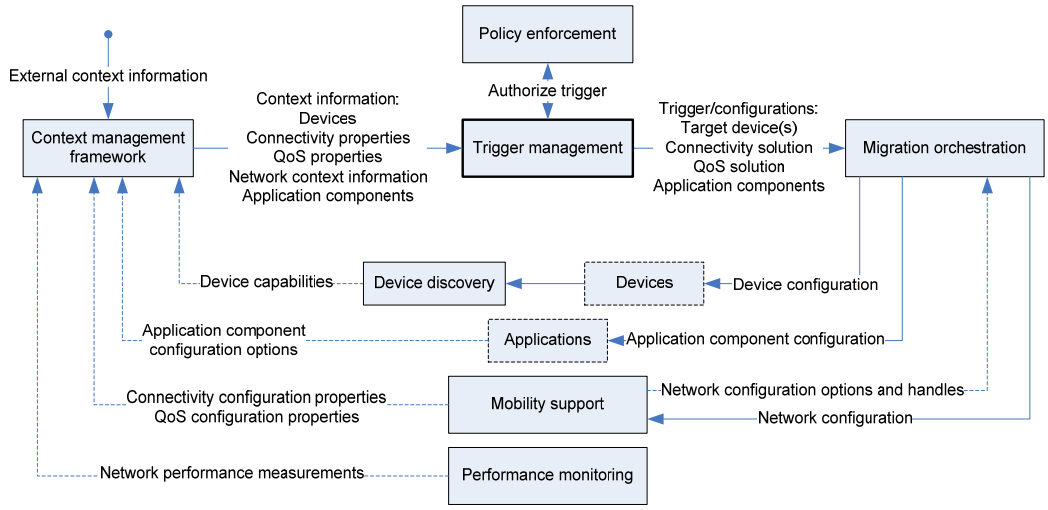

Figure 21 Interactions between Trigger Managment and other components

References to Requirement no 89, 13, 82, 86 and 37.

8.2. CONTEXT MANAGEMENT

The context management system generally handles information distributed in the network and provides easy access for services, application and networking components to their needed information. Thus it is responsible for collecting, storing, processing and delivering relevant information from different sources of context to functions in need of information. As described earlier, context refers both to raw sensor data, or higher level, processed context. Context providers can cover anything from environment changes (the user is standing in front of the device) to very device specific information, such as the remaining battery life. In the OPEN context it is used for many purposes as envisioned and required in the Appendix of D1.1. For example, it is used in the trigger management function that is capable of issuing a migration trigger based on a context change (e.g. once the battery capacity is below a certain threshold). In addition, application logic reconfiguration, UI migration and mobility (see e.g. requirement no. 35 in D1.1) functional modules also require use of context and thus access to it.

See Figure 21 for an example of some of the context information of relevance, as well as for a schematic of how context information flows through components of the platform.

The context management function must be capable of collecting information from a wide range of sources, both on terminal devices and sensors and in the network (e.g. the user's presence status, or information collected from social networks such as Facebook, Twitter, etc.). To accommodate this, the context management function should support distributed collection of information and either distributed or centralized processing of the information.

Context source look up and data retrieval is achieved using a semantic query language, supported by a context broker function and the rest of the management framework.

At present existing solution related to context management have been researched for some time, and in particular a solution originating from the MAGNET Beyond project was investigated and chosen as the basis for the OPEN project, because it is relatively mature and partners have experience using it. The framework is called the Secure Context Management Framework (SCMF) and is presented in [MBD2.3.1], [Bauer06], [Sanchez06] and [MBD2.3.2]. This framework is being extended for the purposes of the project.

Referencing to Requirement no. 65, 89, 135, 136, 34 and 59

8.3. ORCHESTRATOR: MIGRATION ORCHESTRATION

Migration orchestration, also called the Orchestrator, controls the migration process – from received trigger to successful continuation of the migrated service. Migration is triggered either manually by the user or by *Trigger Management*. In either case, the triggering actor tells the *Orchestrator* which application parts to migrate, and where to migrate them. The *Orchestrator* decides how to make migration happen. An example of migration orchestration is when migrating a user interface: then various migration services need to be accessed: the reverse engineering for building the logical description; the UI Adaptation service; the State Handler, which maps the state of the source user interface to the target one; the user interface generator for the target device; and the target device itself for uploading the migrated user interface.

References to Requirement no 6, 13, 82, 30, 48, 51, 79, 80, 139 and 126,

8.4. SESSION MANAGEMENT

Ongoing networking, security or application sessions must be respected when performing migration to provide continuous services.

Examples of such sessions are

- application session: logged in on website, a stateful server (http session and corresponding timeout), service/device registration (e.g. from UPnP)
- security session: established security association (authentication, authorization, credential/key exchange)
- network session: NAT connections, stateful firewalls, TCP connections, multicast group assignment

The session management function helps ensure that sessions can continue after migration.

References to Requirement no 118, 119, 126, 43, 77, 45, 54 and 64.

8.5. POLICY MANAGEMENT AND ENFORCEMENT

A policy management function, which includes policy enforcement, enables users to specify and manage rules about allowing or disallowing migration under certain conditions.

The policy enforcement function oversees conformance to rules which disallow migration based on requirements, e.g., from a user or provider or 3rd party. Conversely it enforces rules that a migration must be effectuated in a given situation. Both types of policy are typically based on contextual calculations, based on functionality requirements, application context, user context and profile, security parameters, etc.

References to Requirement no 142, 66, 49, 52 and 98.

8.6. SECURITY

In several scenarios, security issues arise. Problem such as ensuring privacy protection, prohibiting eavesdropping on wireless transmissions or preventing malicious interactions during the migration process must be addressed. The security function handles support for secure communication by managing keys and ensuring authenticity of participating entities.

If the application uses encrypted channels for communication, the existence of these must also be assured during migration, so that secure service is not interrupted unintendedly.

References to Requirement no 38, 15, 23 and 118.

8.7. PERFORMANCE MONITORING

This component is aimed at monitoring the performance of entities that are involved in the migration environment, because such performance could possibly affect the migration process. Indeed, as network performance, device performance and similar could potentially be important context metrics for migration triggers, performance monitoring functions are needed as context providers for the context management system. Therefore, this module provides input to the Context Management component, Lower-level system performance such as the efficiency of multicore utilization or high-level application QoS metrics are also potential inputs for a performance monitor.

See Figure 21 for a schematic of how this component interacts with others.

References to Requirement no 21, 18, 32 and 102.

8.8. CLOCK/FLOW SYNCHRONIZATION

Different streams, existing or new, need to be synchronized in order to provide continuous service when migrating/distributing an application. An example is enhancing an audio call with video capabilities. When establishing the video stream, this should be synchronized to the audio stream already running.

To apply such synchronization some sort of clock synchronization must be in effect. Also, if the application state is dependent on time, synchronized time between involved devices must be present in the migration process.

References to Requirement no 145 and 58.

8.9. MOBILITY SUPPORT

Mobility support is required in the OPEN middleware to handle network connectivity when the user is mobile so that correspondent nodes do not need to be aware of and handle mobility. Correspondent nodes are application node(s) in remote networks. As these could be non-OPEN-aware application peers, such as regular web- and application servers which are relevant in web-service scenarios and applications, it is not feasible to expect that they can handle mobility themselves. Thus, to support users moving around, e.g. from one network to another or from a fixed infrastructure network to an ad-hoc network, the functionality must be embedded in the OPEN platform and it must provide transparent mobility support to non-OPEN service providers. Mobility support also handles the general case of network reconfiguration, which occurs when an application reconfigures or changes which networks it uses based on changes in context.

Several types of mobility support are considered [Sch00] and [Pop03]:

- *Session mobility* refers to the user's ability to maintain an active session while switching between terminals. For example, a user may join a multi-party teleconference session using his PC while at work. When he leaves to go home, he can switch the session to his mobile phone without interruption, and can continue participating in the conference during his walk

home. Session mobility is very relevant to OPEN, and requires MSP functionality above the network level, e.g., the state persistence functionality.

- *Terminal mobility* refers to the user's ability to take his terminal, move across heterogeneous networks, and continue to have access to the same set of subscribed services, e.g., correspondent nodes. Terminal mobility considers the entire device, and not only one application (or even just one application flow). Therefore, the existing mechanisms supporting terminal mobility may only be applicable for OPEN when the migration application is the only application on the device, i.e. there is a one-to-one mapping between device and application.

- *Personal mobility* refers to the user's ability to transparently access services from anywhere, at anytime, using any terminal. Transparency is enabled by the fact that the user is able to use a single personal identifier on all occasions, regardless of the terminals used or their network locations. This could be used by OPEN, but is not a major concern for the project.

References to Requirement no 104, 35 and 91.

8.10. DEVICE DISCOVERY

The main aspect in migrating services and applications is enriching user-experience. This is done by utilizing additional resources and/or devices available. To use these additional entities, their presence need to be discovered and their capabilities established. This is handled by the device discovery function.

Discovery can be performed in several manners; locally (short-range communication) or centrally (using a commonly accessible registry). The objectives for discovery are also multiple:

- device discovery (network presence)
- service discovery (what services are provided by a device)
- resource discovery (which resources, e.g. battery lifetime, processing power, storage capabilities, etc. are offered by a device).

Regarding the communications with other modules, see Figure 21, which shows how Device Discovery can store information about devices as context, which is managed by Context Management.

References to Requirement no 20 and 33.

8.11. SERVICE ENABLERS INTERFACE

The OPEN Platform and migratory applications access network capabilities through a Service Enablers Interface. This interface provides adapters for the different types of network elements that have to be accessed. The network architecture and the low level interfaces are hidden,

abstracting the underlying layer and providing common functionalities to OPEN middleware and to all the applications that use the middleware.

Service Enablers can be resources external to the OPEN platform providing some network capabilities that can be accessed by the OPEN platform using the Service Enablers Interface component. Such resources might be:

- Presence system where customers and services or applications can publish their presence status or verify the presence status of their buddies
- Location, through which the location information of SIM cards is accessed
- Network Context and access layer information such as:
    - type of network in use (in terms of GSM, GPRS, UMTS, WLAN, Bluetooth, …),
    - type of service in use
    - information regarding the URL accessed by the customer.

The Service Enabler Interface performs privacy related functionalities enabling or denying access according to general privacy rules, customers preferences and profiles.

Information about Service Enablers is stored and managed as context by Context Management.

References to Requirements are indirectly through many other subsystems, since e.g. location is most often provided through such an interface.

9. INTRODUCTION TO A FEW SCENARIOS

This section uses two scenarios to illustrate in more detail some basic interactions between users and components of the platform. The first scenario concerns a migration which is triggered by a change in context. The second concerns a migration initiated by a user.

9.1. SCENARIO: CONTEXT CHANGE INITIATES MIGRATION

Figure 22 illustrates the interactions between a user and some platform components during a context change and a subsequent issue of a migration trigger.

The sequence leads to a migration trigger. The scenario includes the situation in which the user is moving around. The user part considers not only the actual user behavior but also how it is detected through some technology, e.g. a user localization infrastructure (through GPS or RFIDs or something else). The user movement has an impact on the devices available for migration because such devices should be nearby the current user location. Thus, the user movement event should be sent to the device discovery, which will update the current list of devices and services available for migration and send such information to the context management, which can then send the event of context change to the trigger manager.

Thus, the user mobility implies a change in the set of devices available for migration. This change is detected by the device discovery, which is continuously active, communicating with the context manager, which is the entity that generates high-level events for the trigger management, continuously calculating if the context change will result in a migration trigger, as happens in the figure.

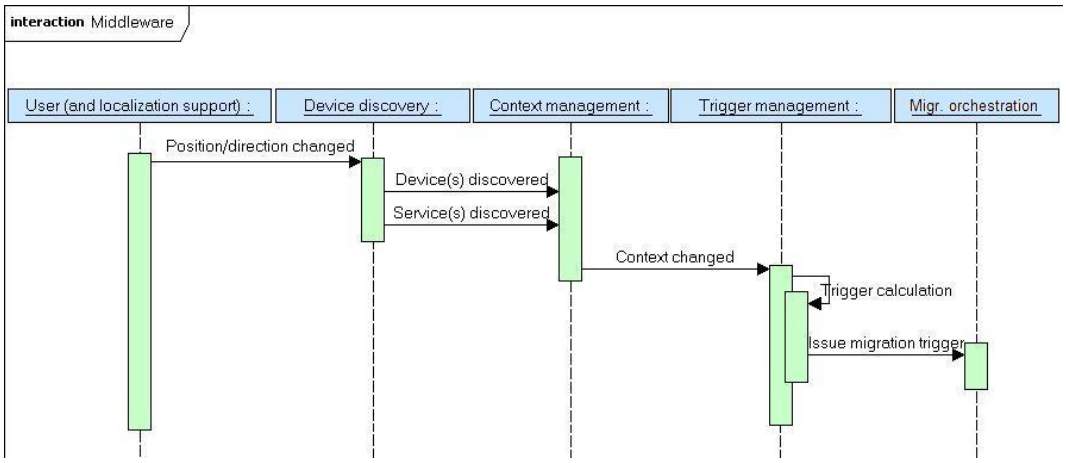

Figure 22 Example interactions between a user and platform components on context change and migration trigger

9.1.1. PERFORMING MIGRATION

The second example, depicted in Figure 24 illustrates the interactions between middleware components during the actual migration, i.e. reaction to a migration trigger. This trigger may be automatically inferred by the system or manually issued by the user, as previously described.

The sequence is initiated upon the migration control function receiving a migration trigger from the trigger management function. The migration control function uses a set of support functions to carry out the migration. In this example the function handling migration policies, security, sessions and network are used during the migration to allow for the application flows to be migrated. As a later step in the migration, several communication/message flows of a running application might be in need of synchronization. This is illustrated by the Sync function (referring to the clock/flow synchronization component. The orchestration of the applications and the various adaptation layer components (represented by *upper layers* in Figure 24) is not included in this figure, which focuses on the supporting functions. Descriptions of all of the supporting functions can be found in Section 8.

9.1.2. PERFORMING APPLICATION LOGIC RECONFIGURATION

In addition to migration, a trigger can also initiate application logic reconfiguration. This includes adaptation to available resources, user needs and other aspects. Figure 23 illustrates the interaction between involved platform components during the adaptation and reconfiguration phase. Depending on the application, different kinds of trigger are needed. Therefore, the reconfiguration control unit registers for specific events at the trigger management component specific for each running application. The reconfiguration control also has to be aware of available component instances which can be used for reconfiguration, e.g. for deciding which of two instances of a component to use in the current situation. Therefore, it registers for such events at the component repository. These events can also initiate reconfiguration and adaptation of the application. The use of both types of adaptation triggers is shown in Figure 23. In both cases, the reconfiguration control (see Section 7) may change connections between component instances, replace instances by other more appropriate instances in the current situation or reconfigure single component instances by adapting their behavior to the current usage context. The gathering of context is done by accessing the context manager during the reconfiguration and adaptation phase. This context information is then used to find the best configuration and adaptation actions in the current situation.

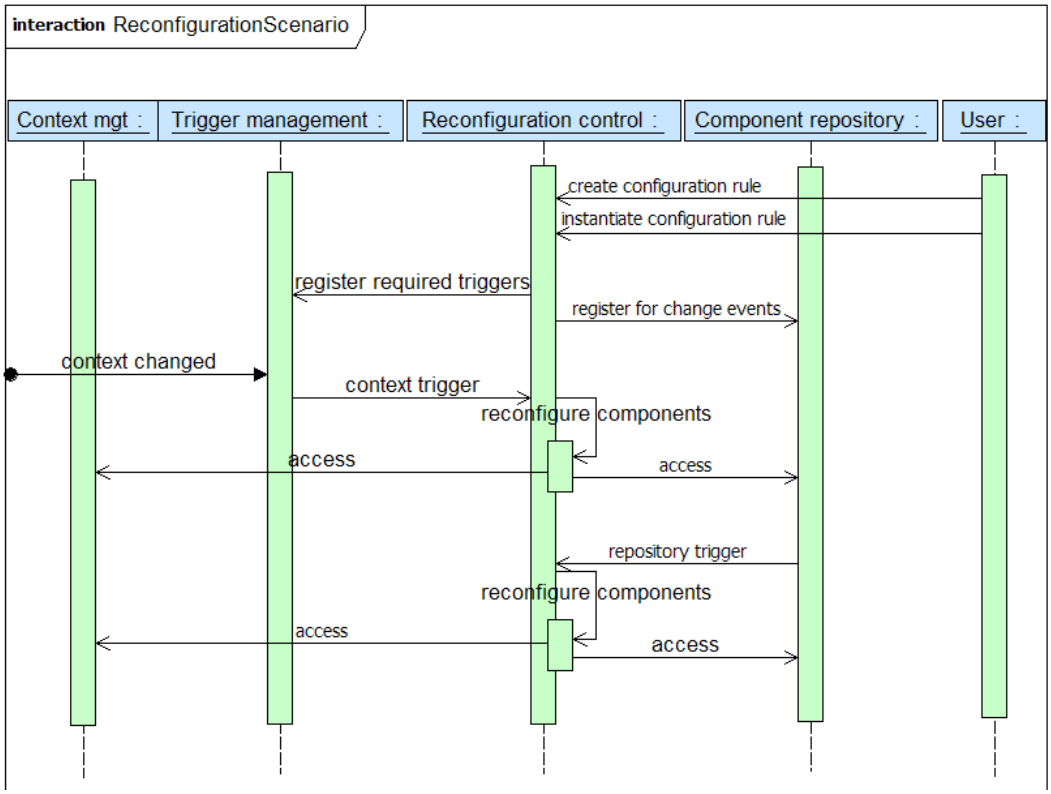

Figure 23 Example of interaction between reconfiguration components and the trigger management for triggering logic/application reconfiguration

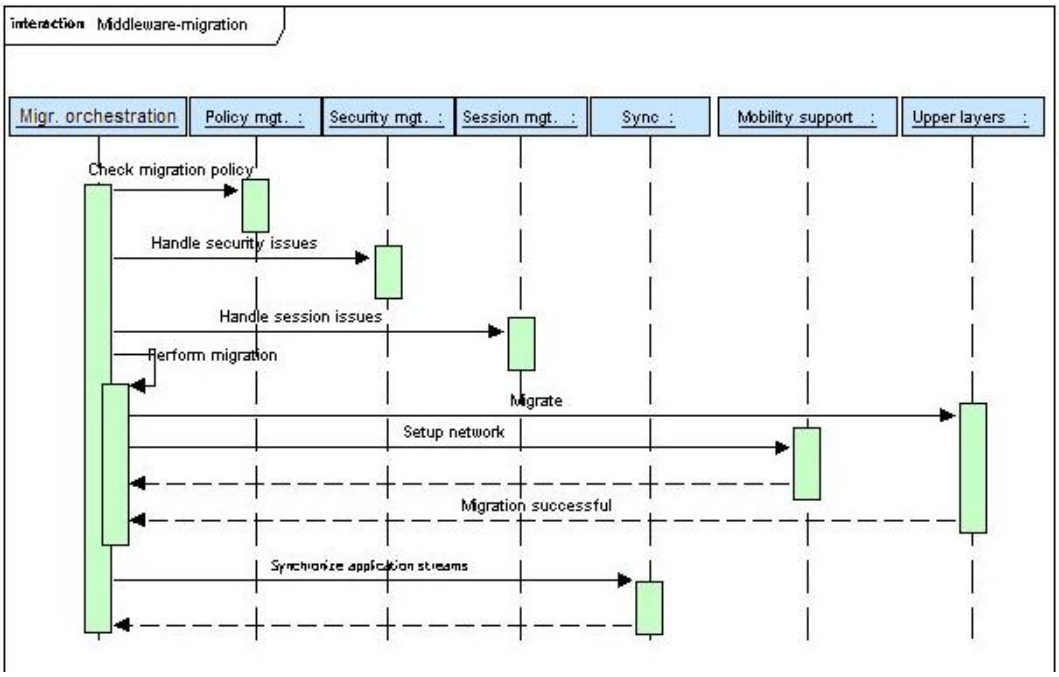

Figure 24 Example interaction between middleware functions during migration

9.2. SCENARIO: USER INITIATES MIGRATION

In this section we analyze a scenario in which a user is interacting with a Web application through a browser on a computer (PC). Differently from the first scenario, in this case we consider a migration triggered by an explicit request from the user.

In this scenario the application is migrated from the PC to a PDA through the use of a migration server, which is one of the possible solutions for the OPEN Platform (as it has been highlighted in Figure 25) and the one currently implemented in the migration prototype developed at ISTI. However, further developments of the prototype could also foresee other solutions, different from the server-based one. In any case, it is worth pointing out that this is an example of a networking-based migration scenario, and that further networking scenarios are elaborated in D3.1.

At first, there is a device discovery phase, which allows the migration environment to identify the devices available for migration. If we consider the access to a web application from a desktop system, such an access could go through an intermediate server, which also has proxy capabilities. Before passing the requested Web page to the client, the server includes, within the page retrieved from the Application Server, a JavaScript excerpt whose purpose is accessing the state of the page at migration time, so that such a state can be preserved for later use. In order to be migration-enabled, the devices should run a migration client, which is used for supporting the device discovery phase (through such a thin client the devices announce their presence to the others) and allow the user to select the target device from the list of devices currently available and then trigger migration. When the migration is triggered, the script inserted sends the state of the Web application, which depends on the user interactions to the migration server. Then, the migration server creates an adapted version for the target device (a PDA in our scenario), associates the state to the corresponding elements of the target version, generates the corresponding implementation and activates it at the point the user left off in the source device.

In the sequence diagram shown in Figure 25 we depict the communication flows occurring between various migration components during a Desktop to PDA Migration. The process has been divided into different phases, distinguished into ”Service Use” (which refers to a normal use of the service by the user), ”Pre Migration” (which are all the operations that have to be carried out for preparing the actual migration), ”Migration” (the step in which the migration is actually carried out), ”Post-Migration” (all the actions that might be done just after migration, before coming back to a normal use of the service).

The scenario starts with a user who is currently using a service through a desktop platform. The module dedicated to discovering the devices available in the current environment and to update the list of devices accordingly together with information regarding their characteristics (the ”Discovery mgt” in Figure 25), discovers in this case a desktop and a PDA. At a certain point the user asks for migration, then a migration trigger is sent by the currently used platform (desktop) to the ”Migration Mgt” module. Then, the currently saved state is sent to the migration server, which also asks to the Discovery module information about the capabilities of the target device involved in the migration. Afterwards, the reconfiguration of the application logic, the adaptation of the user interface to the new device and the association of the state to the adapted interactive

application are carried out, and the reconfigured result is sent to the target device (the PDA in this case) to be loaded. Then, in the Post Migration phase there is a step in which the application device currently running in the source device is terminated. Afterwards the user continues the interaction with the service through the new device.

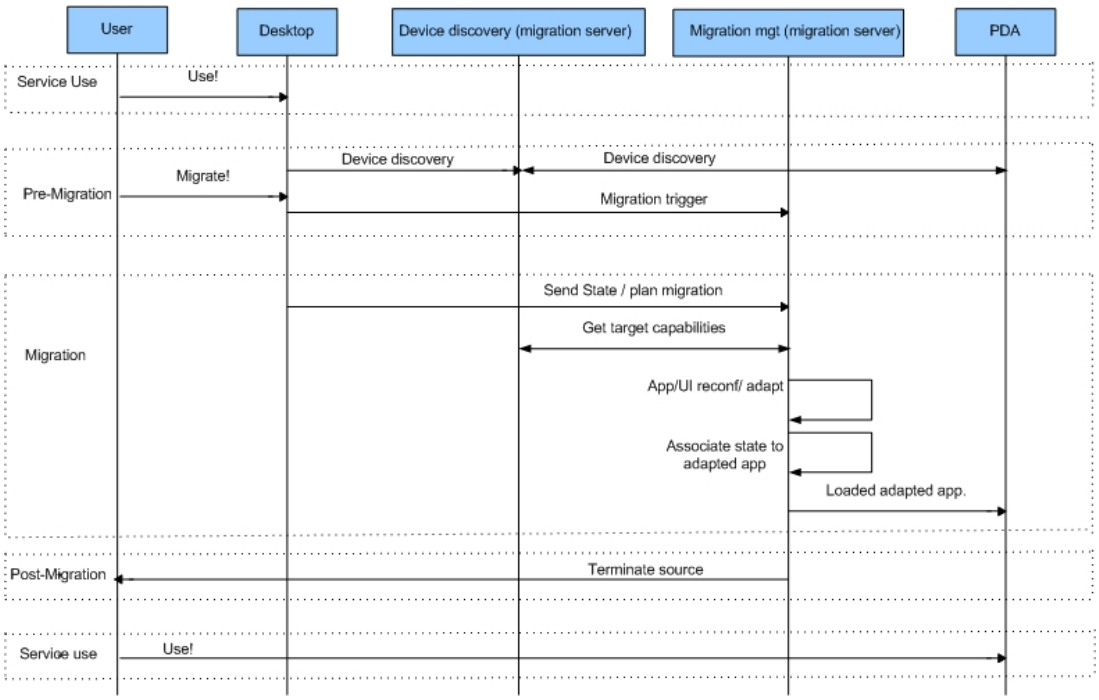

Figure 25 Sequence diagram describing the interaction between middleware migration functions for device discovery, migration triggering, state preservation and application logic reconfiguration

10. CONCLUSIONS AND FUTURE WORK

In this deliverable we presented our revised, and final, version of the OPEN Service Platform architectural framework. An MSP platform, based on this framework, enables the development and execution of migratory applications, which can have three primary capabilities: user interface migration, application logic reconfiguration and network reconfiguration.

Architectural work in Open was led by WP1. An initial platform architecture was developed in D1.2, and its final version is presented here in D1.4. In parallel, WP4 is coordinating the actual implementation of the architecture, as documented in deliverable D4.2, and in the future in D4.4. In this line, the architectural views of WP4 can be seen as development snapshots of the WP1 work. Figure 26 illustrates the timeline for this cooperation, in which D4.4 is the next step.

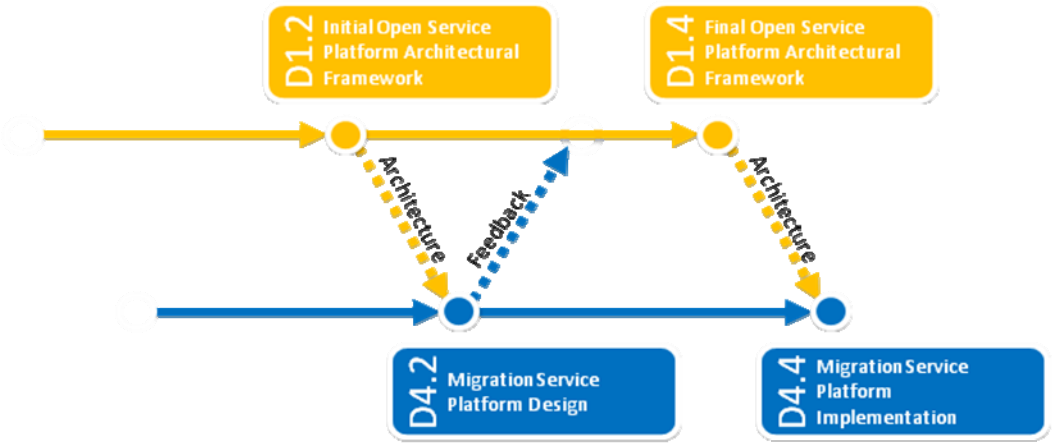

Figure 26 Cooperation between WP1 and WP4

11. BIBLIOGRAPHY

[GSS+02] Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P. Project Aura: Toward Distraction-Free Pervasive Computing. IEEE Pervasive Computing, Vol 21, No 2 (April-June 2002), 22-31.

[PSS08] F. Paternò, C. Santoro, and A. Scorcia, Preserving Rich User Interface State in Web Applications Across Various Platforms, Proceedings EIS'08, Springer Verlag.

[Dey00] A. K. Dey, "Providing Architectural Support for Building Context-Aware Applications", PhD thesis, Georgia Inst. Tech., USA, Nov. 2000.

[OPEN D1.1] Requirements for OPEN Service Platform, OPEN Consortium, June 2008, Faatz, A. and Goertz, M. (eds.)

[OPEN D1.2] Initial OPEN Service Platform architectural framework., 2008

[OPEN D1.3] Final Requirements for OPEN Service Platform, 2009

[OPEN D2.1] Early infrastructure for migratory interfaces, 2008

[OPEN D2.2] Document about architecture for migratory user interfaces, 2008

[OPEN D3.1] Detailed network architecture, OPEN consortium, Feb 2009

[OPEN D4.1] Solutions for application logic reconfiguration. 2009.

[OPEN D4.2] Migration Service Platform Design, 2009.

[OPEN D5.1] Initial application requirements and design. 2008.

[MBD2.3.1] Martin Jacobsen, et.al., Specification of PN networking and security components, Deliverable 2.3.1, December 2006, MAGNET Beyond, IST-027396

[MBD2.3.2] Martin Jacobsen, et.al., PN secure networking frameworks, solutions and performance, Deliverable 2.3.2, June 2008, MAGNET Beyond, IST-027396

[NKA+07] D. Niebuhr, H. Klus, M. Anastasopoulos, J. Koch, O. Weiß, A. Rausch. "DAiSI – Dynamic Adaptive System Infrastructure". IESE-Report No. 051.07/E, Fraunhofer Institut für Experimentelles Software Engineering, 2007

[Bauer06] M. Bauer, R.L.Olsen, M. Jacobssen L. Sanchez, J. Lanza, M. Imine, N. Prasad, Context Management Framework for MAGNET Beyond, IST Summit 2006, Mykonos, Greece, 2006

[Sanchez06] L. Sanchez, J. Lanza, M. Bauer, R. Olsen, M. Girod-Genet, "A Generic Context Management Framework for Personal Networking Environments", Proceedings from 1st Workshop on Personalized Networks, San Jose (CA), July 2006

[Sch00] H Schulzrinne, E Wedlund, Application-layer mobility using SIP, ACM SIGMOBILE Mobile Computing and Communications Review, 2000

[Pop03] Popescu, I., 2003, “Supporting Multimedia Session Mobility using SIP,” in Proceedings of Communication Network and Services Research (CNSR) Conference, May, 2003.

[UML] Object Management Group. UML 2.1 Superstructure and Infrastructure Specifications. November 2007.