# OPEN Project

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

| | |
|---|---|
| **Title of Document:** | Migration Service Platform Design |
| **Editor(s):** | Miquel Martin |
| **Affiliation(s):** | NEC Europe |
| **Contributor(s):** | All Open Partners |
| **Affiliation(s):** | All Open Partners |
| **Date of Document:** | April, 2009 |
| **OPEN Document:** | D4.2 |
| **Distribution:** | EU |
| **Keyword List:** | integration, architecture, design, migration platform, reference |
| **Version:** | 1.0 |

**OPEN Partners:**

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

"The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2008 by Arcadia Design, Clausthal, CNR, Vodafone."

## ABSTRACT

This document provides a self-contained reference of the Migration Service Platform. In the first sections, we deal with the integrated platform design and functionality, the possibilities offered to application developers, and the available operations and interfaces.

This is followed by detailed descriptions of the components and their interactions, and completed with detailed examples of migrations of the project's chosen applications.

The document is intended both as an introduction and a reference guide to the integrated migration platform, and will be further updated as the project progresses.

## TABLE OF CONTENTS


**1. INTRODUCTION .......... 4**

**2. OPEN PLATFORM OVERVIEW .......... 5**

**3. MAKING APPLICATIONS OPEN-AWARE .......... 7**

3.1. THE OPEN ADAPTORS .......... 7
3.2. CLIENT AND SERVER SIDE APPLICATIONS .......... 8
3.3. PARTIAL MIGRATION .......... 9
3.4. APPLICATION-CLIENT SPECIFIC ADAPTORS .......... 10

**4. PLATFORM COMMUNICATION: THE OPEN DISPATCHERS .......... 11**

4.1. COMMUNICATION MODELS .......... 12

**5. OPEN PLATFORM ARCHITECTURE .......... 14**

**6. OPEN INTERFACES .......... 16**

*6.1.1. Interface Design Philosphy .......... 17*
*6.1.2. Ensuring Data Consistency .......... 17*
*6.1.3. Interface Overview .......... 17*

**7. OPEN PLATFORM COMPONENTS .......... 20**

7.1. OPEN SERVER COMPONENTS .......... 20
*7.1.1. CMF (Context Management Node) .......... 20*
*7.1.2. Migration Orchestration .......... 24*
*7.1.3. State Handler .......... 26*
*7.1.4. Trigger Management .......... 28*
*7.1.5. Policy Enforcement .......... 30*
*7.1.6. Mobility support (Server side) .......... 31*
*7.1.7. UI Adaptation (Web) .......... 35*
*7.1.8. Application logic reconfiguration (ALR) .......... 36*
7.2. OPEN ADAPTORS IN THE OPEN CLIENT .......... 38
*7.2.1. CMF (Context Agent) .......... 38*
*7.2.2. Mobility Support (Client side) .......... 38*
*7.2.3. UI Adaptation for Desktop Applications .......... 40*
*7.2.4. Open Client Daemon with UI .......... 41*
*7.2.5. Migration Orchestrator Client .......... 44*
*7.2.6. Web State Handler (Client Side) .......... 44*
7.3. DISPATCHERS .......... 45
7.4. COMPONENT DEPENDENCY .......... 47

**8. OPEN PLATFORM OPERATION EXAMPLES .......... 49**

8.1. MIGRATION OF A WEB APPLICATION .......... 49
8.2. COMPONENT MIGRATION IN THE SOCIAL GAME .......... 50

9. CONCLUSIONS ..... 52

10. BIBLIOGRAPHY ..... 53

A. APPENDIX: OPEN CLIENT INTERFACE ..... 54

B. APPENDIX: OPEN SERVER INTERFACE ..... 56

C. APPENDIX: OBJECT TYPE DEFINITIONS ..... 60

# 1. INTRODUCTION

This deliverable presents the Migration Service Platform design. It is meant as a living document that evolves as development progresses, and new opportunities and requirements are uncovered.

The version in your hands will be continuously modified until Month 24, where D4.4 will present the final prototypes: at that point, part of the accompanying documentation will be this document, updated to the latest changes and agreements, and frozen as the final result of the Integration task.

This document is meant to be used as both an introduction to application developers, and a detailed reference manual. It is structured as follows:

Section 2 provides an overview of the Open Platform, what a typical deployment looks like, and the functionalities an application may expect. Section 3 is directly targeted at application developers, and provides an introduction to making an application Open-aware. In particular, it contemplates different application models and how they may interact. Section 4 discusses the platform communication mechanisms. Finally, Section 5 puts it all together to give an architectural view of the platform, and finally, Section 6 gives a quick introduction to the Open interfaces.

Starting on Section 7, the material becomes more detailed as the individual components of the platform are presented. Of special interest here are the sequence diagrams that exemplify the interaction among them. Section 8 provides an explanation of a complete migration in the case of a web application and a social game. This provides a good overview of the general operation of the platform.

Finally, Section 9 draws our conclusions, and Appendixes A, B, and C respectively provide the parameter level descriptions of the Open Client and Open Server Interfaces, followed by the Object type definitions.

# 2. OPEN PLATFORM OVERVIEW

The Open platform provides a framework that enables applications to migrate between devices, be it by completely migrating, or by sending some of its parts over to another device. Furthermore, the platform provides the application with the necessary mechanisms to adapt to the new device, including adaptation of network connections, user interfaces, and restoration of the previous state.

From a high-level view, as illustrated in Figure 1, the Open server sees only any number of Open Clients. By using the Open interface, applications can request migrations and adapt to new devices. The Open server, in turn, can request other Clients to execute applications and restore their state.

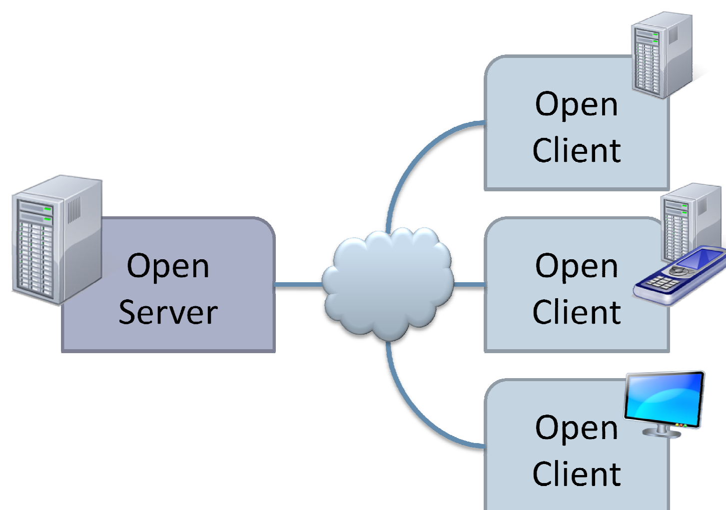


**Figure 1 Overall architecture, where applications are seen as clients of the migration platform**

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 2. 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.

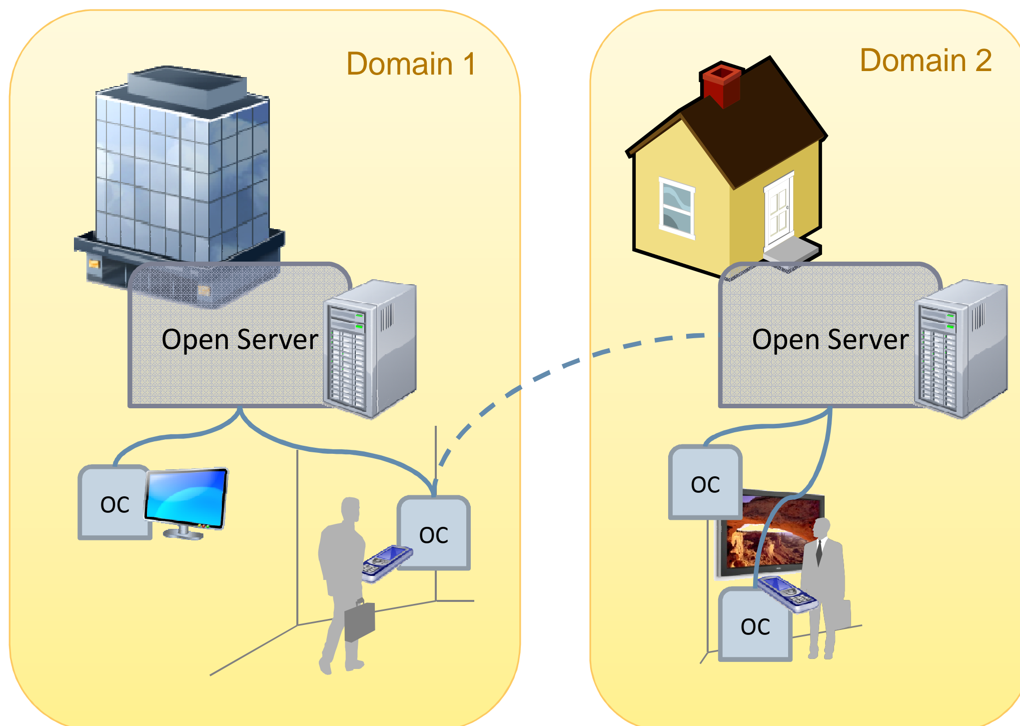


Figure 2 Open Servers may be localized, and clients might handover between them

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.

# 3. MAKING APPLICATIONS OPEN-AWARE

One of the objectives of the Open project is to abstract as much migration functionality as possible from the application, thus making it easy for developers to integrate with our solutions.

An Open Client, which behaves as a Black Box as far as the platform is concerned, is actually composed of a device which runs the applications and calls on the Open functionality.

## 3.1. THE OPEN ADAPTORS

Figure 3 illustrates the internals of an example Open Client. The Client is made in this case of a Terminal (e.g. a mobile phone) which runs a native application and a set of Open Adaptors. The adaptors implement the part of the migration functionalities that are common across applications. They are meant to be reused across applications, so long as the platform permits it. In doing so, an application needs only to make use of the adaptors to become migration capable. Naturally, there is still complexity involved in modifying the application logic to support the new migration lifecycle, but all other common tasks are extracted onto the adaptors.

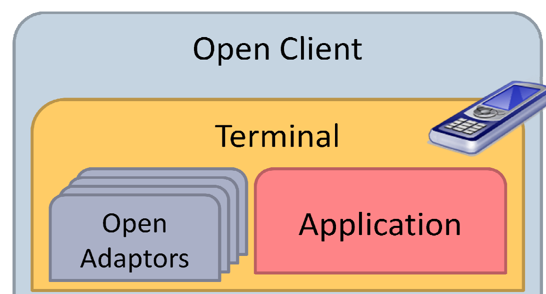


**Figure 3 Applications implement or use the Open Adaptors, and in doing so, present an Open Client interface to the Open platform**

That said, Open Client needs only to present an Open Client interface to the Open Server, so the use of the adaptors is actually optional. As long as applications respect the interface, they may override the default adaptors and re-implement their functionality internally. This is some times necessary to support the certain types of application available nowadays, and especially those found in the web. Consider for instance:

- Web based applications are split between their execution environment (a browser) and their actual content (HTML, JavaScript and embedded objects like Silverlight or Flash)
- One might want to support certain migrations, such as web pages, without modifying the existing web servers. This requires pulling more functionality towards the client

Because of the richness of the application ecosystem, and to provide more flexibility to developers, applications can integrate with Open by using the Open Adaptors exclusively, or by reimplementing them and offering the Open Interface directly from the applications. Any combination in between is just as well possible, where applications reuse some adaptors, and override others.

## 3.2. CLIENT AND SERVER SIDE APPLICATIONS

Another crucial difference between application types is how many of them run on a server, and how many are actually client-based. Whenever state has to be kept, for instance, it might be necessary to harvest state information from both the client and the server. In another example, mobility support might require adaptation at the client side.

The Open approach here is again that of flexibility. Assuming that the *combined* client and server side of an application can behave, as a whole, as an Open Client would, the Open Server will behave as expected.

This simply means that the sum of the methods of the interfaces of the Open Adaptors should add up to implement all the methods of the Open Client interface. If this is the case, the platform will not concern itself with where each of the adaptors is actually running. Figure 4 illustrates this scenario.

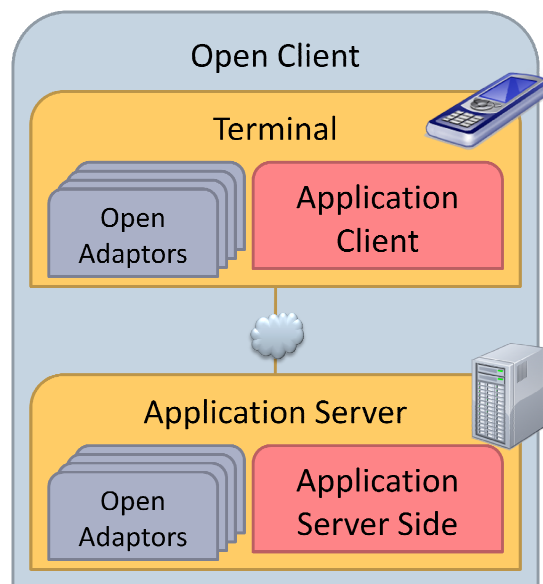


**Figure 4 Applications look like a single Open Client to the platform, even when they have a server and a client part**

We have now described two of the choices that application developers have at their disposal when integrating with the Open Platform. The applications that have been developed so far, and which are currently being integrated, fall in different categories. Figure 5 shows how these applications map to the two parameters proposed.

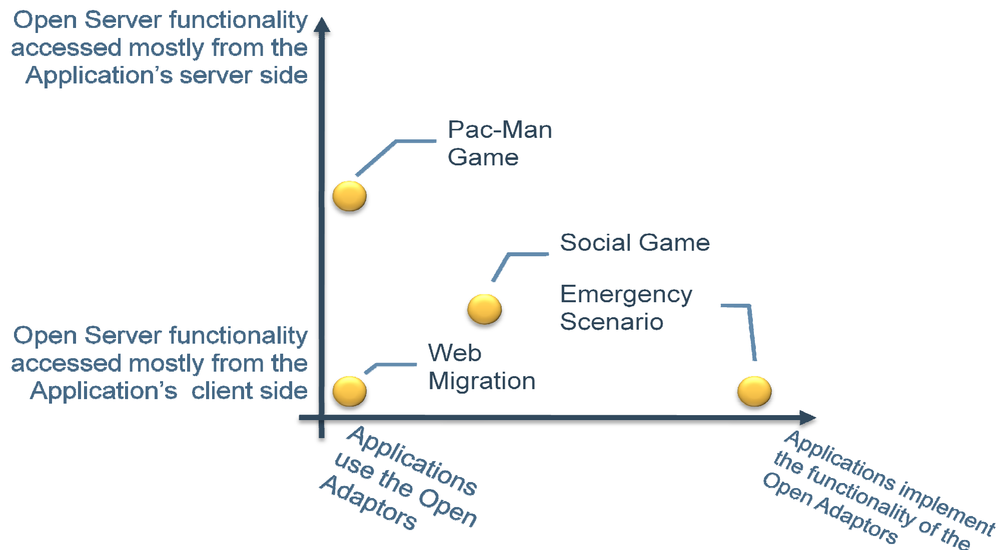


Figure 5 Different applications can choose to adapt differently to the Open platform to achieve migration support

The **Emergency Scenario**, for instance, is a mostly server based, .NET and Silverlight application. As such, it is difficult for it to integrate with all of the available Open Adaptors, and therefore implements most of that functionality itself, from the server side.

The **Web Migration** applications, however, makes a point of requiring no changes from the web server, and so, orchestrates migrations using the terminal and the Open Server (where a Web UI adaptation proxy is running). Also, it makes full use of the available Open Adaptors.

The **Pac-Man** game makes full use of the Open Adaptors, especially those for Application Logic Reconfiguration, and has some components on the server side.

The **Social Game**, finally, makes use of most Open Adaptors, and uses a homogeneous mix of client and server side components.

## 3.3. PARTIAL MIGRATION

The Social Game, additionally, illustrates the concept of partial migration, where an application distributes itself across multiple devices. Technically, this can be achieved by running the different application components in different devices, and then either

- Presenting themselves to the platform as individual Open Clients
- Running the components, logically, as a single Open Client

The Open Platform will consider each Open Client as an application, so the choice in deployment will influence the way applications are handled. In the first case, the application components will have to manage their own relationship, while in the latter, the Open Server is aware of the connection between the components that run in the same Open Client.

## 3.4. APPLICATION-CLIENT SPECIFIC ADAPTORS

Certain specific adaptors are, by nature, not suited for the server side of an application. This is the case of the GUI Adaptation for java programs. The Namuco toolkit (see section 7.2.3) offers multicore support for GUIs, and these are obviously bound to the terminals, and make sense only on the client side.

Additionally, because of its low level nature, the link to these Open Adaptors is made available to Java applications or runtime modules by linking to the Java part of the Namuco library, i.e., by including the corresponding package.

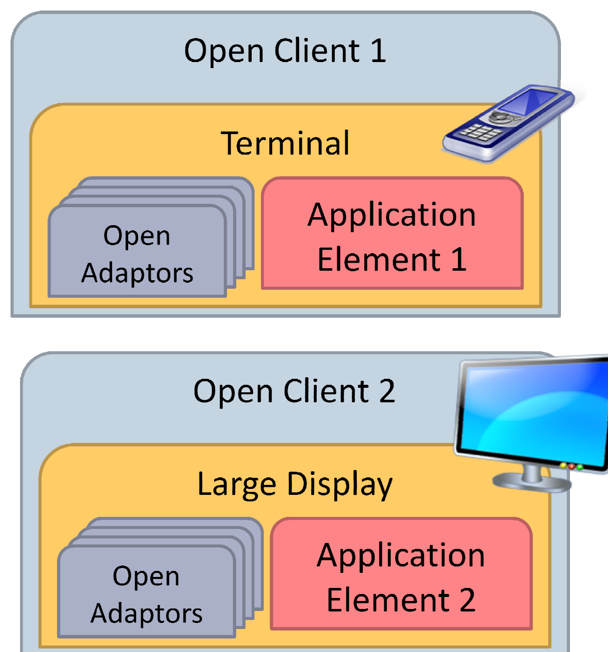


**Figure 6 An application that has partially migrated, and presents itself as two Open Clients**

# 4. PLATFORM COMMUNICATION: THE OPEN DISPATCHERS

The Open Platform is built on the principle of modularity. The Open Server provides its functionality through multiple Server Components. The Open Client is made up of a number of Adaptors, and an application, which can be further split between a server and a client side, and then again into components.

While this makes the Open Platform easier to customize to the specific needs of both the deployment devices and the integrating applications, it comes at a price. Consider, for instance, the Open Server Interface, which is made up by the sum of the interfaces of the Open Server components. How would an Open Client know which component to address, when a certain part of the interface was to be invoked?

It is clear that communication hubs are required that factor out this complexity and provide some sort of directory service.

The proposed solution in Open follows the dispatcher pattern, where a Dispatcher module acts as a single endpoint for a number of components and, at the same time, those components use the dispatcher to route their messages. Figure 7 shows how this concept is applied to the Open Platform:

**Figure 7 Dispatchers in the Open Server and Client act as a communication hub**

Notice how the highlighted dispatchers funnel the communication of the multiple components. We can identify three types of Dispatchers:

- **The Open Server Dispatcher:** is unique for each Open deployment, and it is the endpoint for the whole Open Server. Any Open Client that needs to make a request to the server will do so through this Dispatcher, and all Open Server components will send their messages to the Open Clients through it.

- **The Open Client Dispatcher:** is present once per Open Client, and it represents the unique contact point through which the Open Server can send messages to the Applications and Adaptors it contains. Likewise, when an Open Client component needs to send a request to the Open Server, it does so through the Dispatcher. Note how, should an application be split between its server and client sides, the Open Client Dispatcher takes cares of sending the message where it is needed.
- **The Open Adaptor Dispatchers:** are bound to a specific application/device pair. They are meant to provide the whole functionality of the platform to the applications in a compact and easy way.

As in the case of the Adaptors, applications can choose to use the existing dispatchers, or implement their own. Whichever the case, the Dispatcher must fulfill its functions, including the aggregation of the interfaces of the individual components, so that:

- Applications see an Open Server Interface provided by the Adaptor Dispatcher
- Adaptors see an Open Client Interface provided by the Adaptor Dispatcher
- Open Servers see an Open Client Interface provided by the Open Client Dispatcher
- Open Clients see an Open Server Interface provided by the Open Client Dispatcher.

These interfaces are consistent with the directional view presented in Figure 10 (page 16).

In using the dispatcher infrastructure, applications need to communicate only with a statically configured Dispatcher, and need not worry about the underlying network conditions. This is true to the point that components can now deal with ComponentIDs alone, while the specific service endpoints, which might change at runtime in mobility scenarios, remain hidden.

## 4.1. COMMUNICATION MODELS

The communication protocol for the Open Platform is XML-RPC. This assumes that components can listen for requests, and even receive asynchronous responses. This, however, can be problematic in at least the following cases:

- **Firewalls and Network Address Translators (NATs):** can block incoming traffic, either through the enforcement of firewall policies, or because NATed components run on a private IP which is not routable from the other communication endpoint.
- **Application platforms which don't support listening modes:** the most clear case is that of web applications, which can make HTTP requests (even asynchronous ones using AJAX) but can not listen to messages initiated by an external peer (with AJAX frameworks like Direct Web Remoting (DWR) or Comet being a notable exception).

To solve this problem, the Open Dispatchers support a polling mode on their server side. Messages that would usually be directly called on the Client Interface are instead buffered at the dispatcher. The client end can then poll periodically to retrieve pending messages. Further details and sequence diagrams can be found in Section 7.3.

**NOT TO FORGET:**

- Only the Open Server interface offers a polling mode for asynchronous execution of the Client side interface
- Because of that, all Open Client methods are asynchronous
- Dispatchers interact with Mobility Support to route Platform communications in mobility scenarios

# 5. OPEN PLATFORM ARCHITECTURE

The aim of this section is to put together all the pieces presented in the previous sections, providing an overview of the Open Platform architecture.

Architectural work in Open is lead by WP1: initial platform architectures were developed in D1.2 (1) and its final version will be presented in D1.4. In parallel, WP4 is coordinating the actual implementation of said architecture, starting in this deliverable D4.2, and once again in D4.4. In this line, the architectural views of WP4 can be seen as development snapshots of the WP1 work. Figure 8 illustrates the timeline for this cooperation.

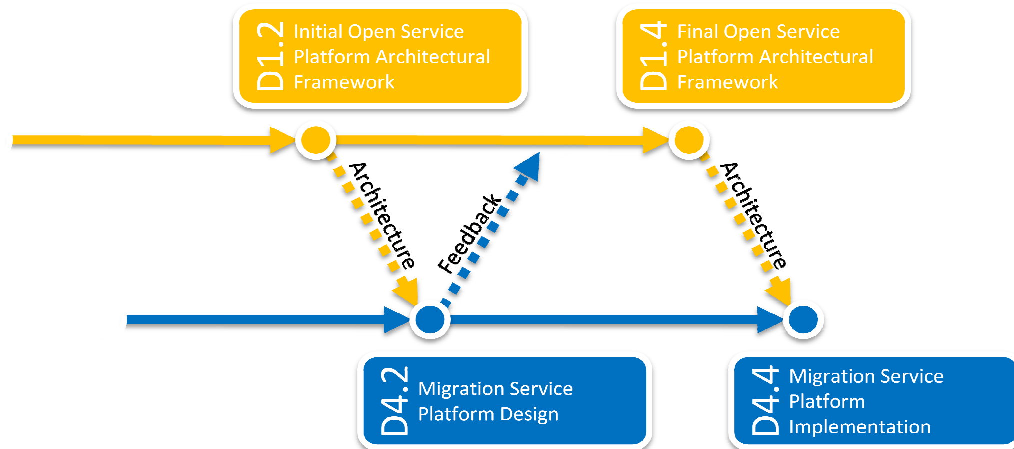


**Figure 8 Cooperation between WP1 and WP4**

With this in mind, we now present the overall architectural picture for D4.2 in Figure 9. A normal deployment will include one Open Server and any number of Open Clients. All communication between the different segments occurs through a Dispatcher, and the application can override any or all of the client side components.

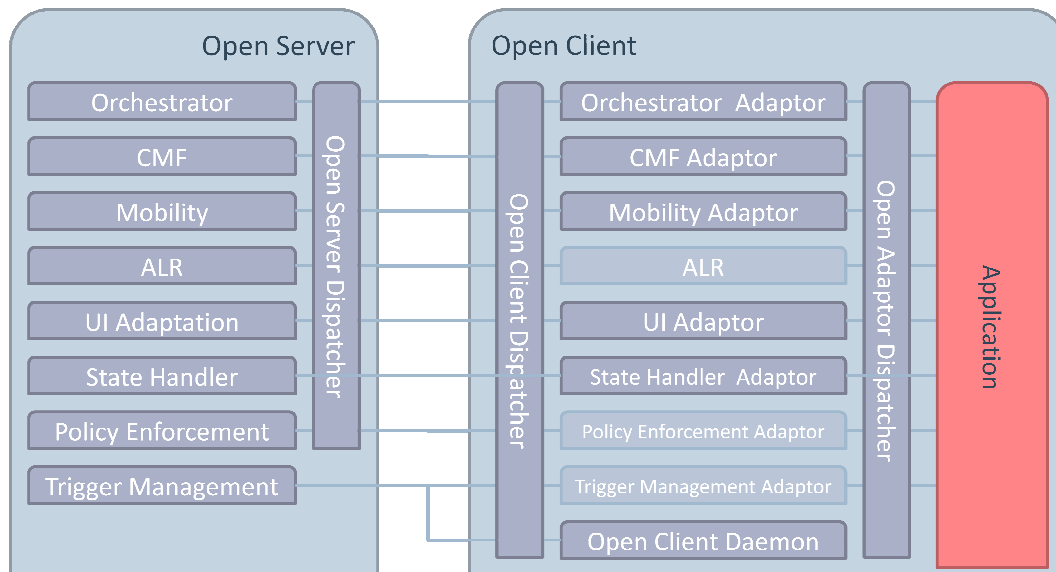


**Figure 9 Open Platform Architecture**

The Application Logic Reconfiguration (ALR), the Policy Enforcement and the Trigger management, do not require an adaptor counterpart, since they do not possess any client side functionality. While they do not exist as a software instance, they do have a logical significance, in that the Application seems to talk to the adaptor component on its way to the server. Because their functionality is limited to relaying messages between the application and the server components, we have chosen to implement this in the dispatcher instead.

Trigger management is a special case in that manual migration triggers can be generated internally in the application, or using the Open Client Daemon; In this sense trigger generation is represented as a link to both the Trigger Management Adaptor and the Open Client Daemon.

The details of each component, the interfaces it offers, and the interactions with the rest of the platform are explained in detail in section 7.1 on page 20 and section 7.2 on page 38.

Finally, we would like to highlight the role of the Orchestrator component. As its name suggests, it is responsible for interacting with the rest of the platform components, and to coordinate the lifecycle of applications as they migrate. Further details can be found in section 7.1.2 on page 24.

**NOT TO FORGET:**

- All Open Platform communication on the wire travels from and to a Dispatcher
- Applications can choose to implement any of the Open Client functionality
- In what regards platform communication, an Open Client will never talk directly to another Open Client: it will go through the Open Server

# 6. OPEN INTERFACES

After several phases, the project has chosen to summarize all its functionality into the Open Client and Server interfaces, both of which are implemented where necessary using XML-RPC.

These interfaces, however, are not limited to the Open Server and Open Client blocks, but throughout the platform:

- The **Open Server presents an Open Server interface** to the Open Client but,
- The **Open Adaptors** also **present an Open Server interface** to the Applications
- Likewise, the **Open Client presents an Open Client interface** to the Open Server, and
- The **Application** also **presents an Open Client interface** to the Adaptors

In other words, the platform uses only two interfaces: which one of them a component implements for its neighbor, and which one it can expect from that neighbor, depends exclusively on the direction we're looking at: towards the server, all components implement the Open Server Interface; towards the client, the Open Client one. Figure 10 illustrates this concept.

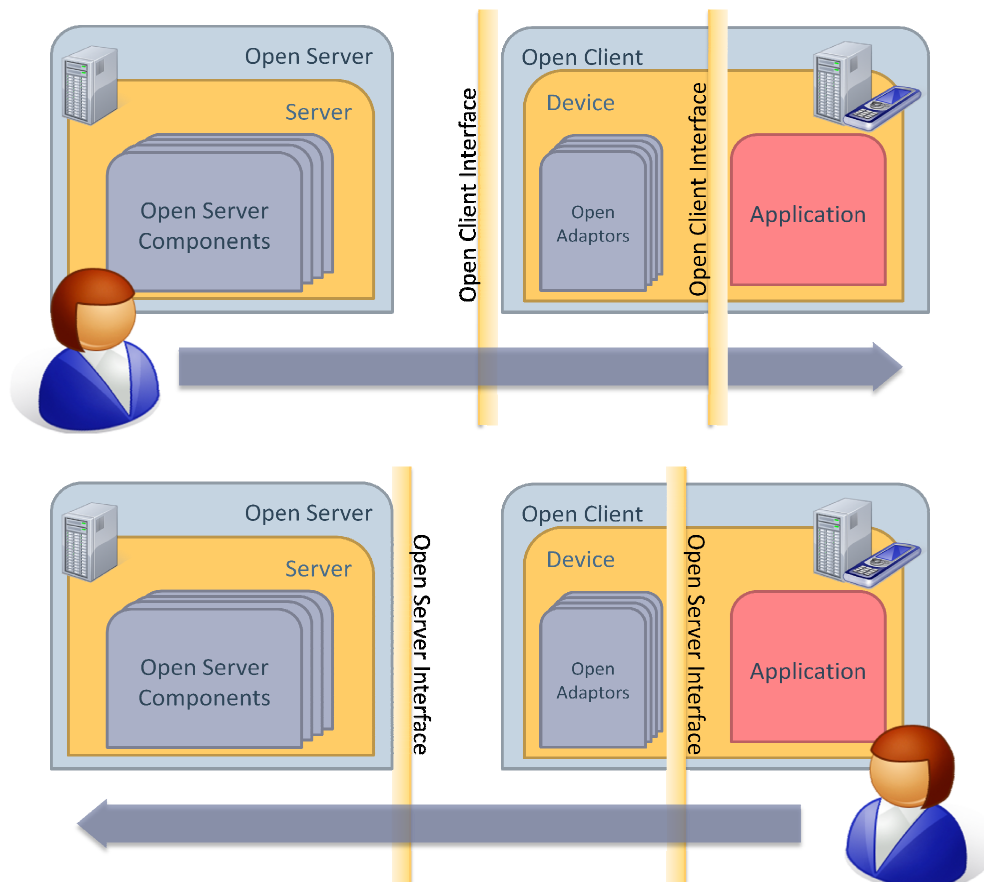


**Figure 10 The interfaces offered to a component depend only on the direction the component is "facing"**

Note that it is up to the application developer whether to override the Open Adaptors, or implement their functionality within the application. Whichever the choice, the aggregated interfaces provided by the Adaptors, and the Application, must always amount to a complete interface. The routing of external calls to the adequate implementing component will be handled by the dispatchers, described in Section 4 and in further detail in Section 7.3.

### 6.1.1. INTERFACE DESIGN PHILOSPHY

The Open Interfaces are implemented using XML-RPC, an application independent Remote Procedure Call mechanism based on XML, which takes a method name and any number of name/value pairs as parameters.

In order to simplify the implementation of the interfaces, we have strived for a minimalistic approach, with no method overloading and a minimal set of parameters. The complete listing of methods can be found in appendixes A and B for the Open Client and Open Server interfaces respectively.

The complexity needed for the platform, therefore, has been transferred to the Data Types that form the parameters. Appendix C has a complete listing of object types.

Because certain clients can not listen for Client interface methods, the Open Dispatchers support a polling mode. This, however, means that the Server components will not get a reply until the next polling cycle: to avoid unnecessary blocks, all Open Client interface methods must be asynchronous. This results in a number of Open Server methods which are in fact the Callback of their Open Client counterparts.

### 6.1.2. ENSURING DATA CONSISTENCY

All running components, applications and a number of other information, is kept at the Open Server. For example each application has an associated Application object, which contains information such as the components that form the application. On occasion, said objects might be sent over to the Open Client, where they might be changed. It is therefore important to keep track of the changes and ensure synchronization of the instances of the object.

To achieve this, we have determined that objects may, in general, be modified only at the Open Server Orchestrator component. When client side processing is required, a method is used to send the object to the Client. Likewise a corresponding callback method receives the modified object. The Open Server must therefore relinquish its lock on the object from the moment it's sent out, until it is returned. In doing so, we can guarantee that the Object will only exist at one place at any given time.

### 6.1.3. INTERFACE OVERVIEW

The following listing provides the complete list of Open Interfaces; Instead of going into details here, we would like to suggest that the reader learns about them in the context of the

component that implements them (see Section 7) or refers to the Appendixes in this document for their parameter level descriptions.

| Open Server Interface methods | |
|---|---|
| Methods at the Orchestrator component and ALR | String deviceID:: registerDevice(Device device) |
| | String applicationID::registerApplication(Application application) |
| | String componentID::registerComponent(Component compenent) |
| | Boolean::unregsiterDevice(String deviceID) |
| | Boolean::unregisterApp(String applicationID) |
| | Boolean::unregisterComponent (String componentID) |
| | Device[]::getDevicesSupporting(String[] componentID) |
| | void::runningStatusSet (String componentID, String runningStatus) |
| | void::ApplicationReconfigured(Boolean isReconfigured, String applicationID) |
| | void::componentReconfigured(Component component) |
| | boolean::triggerMigration(String targetDeviceID, String[] componentID) |
| | void:: retrieveUI(String componentID) |
| | void:: adaptedUISet(Boolean isUpdated, String componentID) |
| | void::stateRetrieved (State state) |
| | void::stateSet(Boolean isSet, String ComponentID) |
| Methods at the Mobility Support component | void::networkReconfigured(String deviceID, boolean isReconfigured) |
| | boolean::registerNetworkConfiguration(String deviceID |

| Open Client Interface methods | |
|---|---|
| Orchestrator component and Client Daemon | void::setRunningStatus (String componentID, String runningStatus) |
| | void::reconfigureApplication (String applicationID, Configuration configuration) |
| | void::migrationTriggerAccepted(String sourceDeviceID, Boolean accepted) |
| | boolean::reconfigureNetwork(String deviceID, NetworkConfiguration networkConfiguration) |
| | void::reconfigureComponent(Component component, ConfigurationInstruction configurationInstruction) |
| | void:: UIRetrieved(String ComponentID, UI ui) |
| | void:: setAdaptedUI(UI ui, String componentID) |
| | void::retrieveState (String componentID) |
| | void::setState (String componentID, State state) |
| Mobility Support component | boolean::reconfigureNetwork(String deviceID, NetworkConfiguration networkConfiguration) |

The following sections will detail the different components of the Open Server and Client.

**NOT TO FORGET:**

- Open Client interfaces look the same regardless of the place where they are being implemented. The functionality of an Open Server component, however, is likely to be different (and probably complementary) to that of, for instance, the Adaptor
- All Open Client methods are asynchronous so they can support the polling mode at the dispatchers.
- Object IDs are unique for an Open deployment
- Objects can only be modified by the Open Server Orchestrator, unless they have been sent out (temporarily) to the client side, in which case the client owns the modification lock.

# 7. OPEN PLATFORM COMPONENTS

## 7.1. OPEN SERVER COMPONENTS

### 7.1.1. CMF (CONTEXT MANAGEMENT NODE)

The Context Management Framework (CMF), described in (2) will provide the components and applications with easy access to context information. The framework, originally developed in the MAGNET Beyond project (3), and extended in OPEN, relies on a set of distributed Context Agents to ensure that context information can be efficiently searched and distributed. Each Context Agent has the capability of collecting local available context information through retrievers (small specific software components that allow the context agent to interface to any arbitrary information source). Furthermore, it also has local storage capability, additional processing of information and the necessary communication means including scoping and other search parameters to locate relevant information for the application. The framework supports both synchronous and asynchronous access models, which are specified via the specific context access query language.

At the server side, a Context Agent with a special role or configuration is located, namely as Context Management Node. This role is special in a sense; it is acting as a central registry repository for all available context information within a specific network domain. Therefore this node has knowledge of all available context information, whereas clients (other Context Agents) may need to ask this node for the location of other information. This role of the Context Agent is setup through a configuration file as a part of the Migration Server configuration.

#### OPEN INTERFACES

The Context Management Framework does not directly implement Open interfaces. It is a support component providing context information, which is accessed using platform internal methods, specially by the Trigger Management and Application Logic Reconfiguration (ALR) components.

#### INTERNAL INTERFACES

The CMF interacts with applications via XML-RPC calls, whereas the XML is describing the used query language CALA (Context Access Language). This language has several basic types of queries:

- Query for information
- Subscribe to information / unsubscribe
    - Periodic update of information with specified update intervals
    - Event driven updates, i.e. whenever the information changes value larger than some specified level

- Insert / Update / Delete information in the CMF storage

Besides these core parts of a query, additional parameters can be needed, namely to 1) scope the query, i.e. how local is the query supposed to be (node local, network local, global, etc..) and 2) if there are any options to the query, this can be specified, e.g. security parameters. An example for a simple request query could be

```
<cala xmlns="http://nle.nec.de/CMF" >
    <query>
        <entityIdentifierSelector>
            <hasIdentifier>Display1</hasIdentifier>
            <entityType>PublicDisplay</entityType>
            <attributeName>nearbyPerson</attributeName>
        </entityIdentifierSelector>
        <scope>
            <networkScope>NODE</networkScope>
        </scope>
    </query>
</cala>
```

At the transport level, XML-RPC is a well known protocol that transports method signatures (including parameters) and their responses using XML over HTTP.

The specific content of the messages is detailed in the CMF Tutorial (4) presented to the consortium in March 2009. For illustration purposes, we provide here a possible response to the prior query:

```
<cala xmlns="http://nle.nec.de/CMF" >
    <queryResponse>
        <entities>
            <entity>
                <hasIdentifier>Display1</hasIdentifier>
                <entityType>PublicDisplay</entityType>
                <attribute>
                    <name>nearbyPerson</name>
                    <type>string</type>
                    <value><string>Laura</string></value>
                <metadata>
                    <name>timestamp</name>
                    <type>MetaData</type>
                    <value><string>1216739890695</string></value>
                </metadata>
            </attribute>
        </entity>
    </entities>
</queryResponse>
</cala>
```

Abstracting the XML-RPC into method signatures, one can see the CMF interface as follows:

| **QueryResponse:Query(Selector, Scope)** | |
|---|---|
| Selector | An xml element detailing at least the type and attributes to be queried |
| Scope | Whether the query is limited to this Context Agent, or also to those attached to it |
| *Returns:* | A QueryResponse object with a list of entities |
| Query for Context Information on a given entity or entity type | |

| **GID::Subscribe(Selector, SubscriptionCondition, Scope)** | |
|---|---|
| Selector | An xml element detailing at least the type and attributes to be queried |
| SubscriptionCondition | Specifies the circumstances that should trigger a notification on new context |
| Scope | Whether the query is limited to this Context Agent, or also to those attached to it |
| *Returns:* | A Global Subscription Identifier which can be used to track the subscription and link to the received notifications |
| Subscribe to context information. | |

| Void::Insert(Entity[], Scope) | |
|---|---|
| Entity[] | A list of entities that need to be inserted |
| Scope | Whether the entities should be inserted in all the nodes or just on this context agent |
| *Returns:* | Void |
| This method is used to insert information into the Storage component of the Context Agent. The information can later be retrieved as if it were standard context | |

| Void::Update(Selector, Attribute[], Scope) | |
|---|---|
| Selector | An xml element detailing at least the type and attributes to be updated |
| Attribute[] | The list of attributes to be updated |
| Scope | Whether the entities to be updated are those in all the nodes or just on this context agent |
| *Returns:* | Void |
| This method updates the attributes of entities already present in the CMF | |

| Void::Delete(Entity[], Scope) | |
|---|---|
| Entity[] | The list of entities to be deleted |
| Scope | Whether the entities to be deleted are those in all the nodes or just on this context agent |
| *Returns:* | Void |
| This method updates the attributes of entities already present in the CMF | |

We would like to refer the reader to (4) for further details; If needed, a dedicated CMF tutorial can be arranged.

## INTERACTIONS

The CMF interacts with many different components in the OPEN system, as it provides general information as needed. The interesting part for the CMF, however, is how this information gets into the system. Here, this is done mainly through retriever components which are small software components responsible for interacting with the data source itself.

Since the CMF also has the capability of providing inferred (non-measurable) context information which may need externally obtained information, the CMF may need to set subscriptions or query to other Context Agents in the network. This will appear as if any other application triggered a CALA query, and hence the configuration of the processing part of the CMF for this purpose is similar to specifying any CALA query.

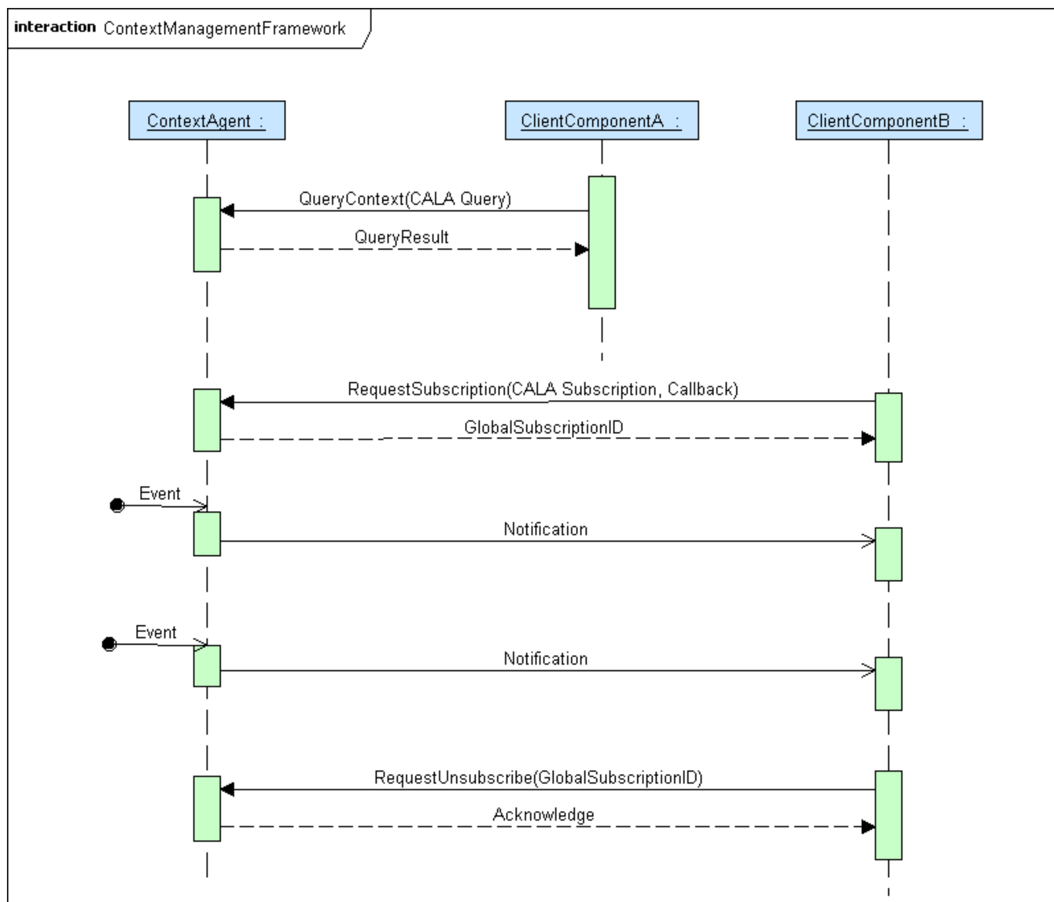


**Figure 11 Different interaction patterns between a Context Agent and different components.**

The different approaches of interaction are as shown in Figure 11, with at the top a Query using an XML-formatted CALA query as parameter as input is shown from ClientComponentA. A response in terms of an <Entity> list is returned.

Following this, an example of subscription interaction from ClientComponentB is shown, which initially consists of a subscription request with the CALA subscription query and callback information as input. A global subscription ID is returned if successful. Then on events (either a timeout for periodic update, or a change in context information for event driven updates) a notification with the latest value is sent to the subscribing component via the callback information provided. Finally, the client component can cancel the subscription by using an unsubscribe call with the global ID provided earlier, and an acknowledge is returned if successfully unsubscribed.

### 7.1.2. MIGRATION ORCHESTRATION

The Migration Orchestration is the module that coordinates the various phases of the migration. It manages two kinds of migration: local applications migration and web applications migrations. Moreover, a support for partial migration is offered.

#### OPEN INTERFACES

The Orchestrator makes up the majority of the Open Server and Client interface methods, since it's the communication hub for all migration operations. Open Components, especially Trigger Management, may use the following Open Server methods to trigger a migration. The following table covers the whole set of methods:

| Open Server methods at the Orchestrator component | String deviceID:: registerDevice(Device device) |
|---|---|
| | String applicationID::registerApplication(Application application) |
| | String componentID::registerComponent(Component compenent) |
| | Boolean::unregsiterDevice(String deviceID) |
| | Boolean::unregisterApp(String applicationID) |
| | Boolean::unregisterComponent (String componentID) |
| | Device[]::getDevicesSupporting(String[] componentID) |
| | void::runningStatusSet (String componentID, String runningStatus) |
| | void::ApplicationReconfigured(Boolean isReconfigured, String applicationID) |
| | void::componentReconfigured(Component component) |
| | boolean::triggerMigration(String targetDeviceID, String[] componentID) |
| | void:: retrieveUI(String componentID) |
| | void:: adaptedUISet(Boolean isUpdated, String componentID) |
| | void::stateRetrieved (State state) |
| | void::stateSet(Boolean isSet, String ComponentID) |

Similarly, applications and Open Client orchestrators may expect the following methods to be called on them:

| Open Client methods at the Orchestrator component | void::setRunningStatus (String componentID, String runningStatus) |
|---|---|
| | void::reconfigureApplication (String applicationID, Configuration configuration) |
| | void::migrationTriggerAccepted(String sourceDeviceID, Boolean accepted) |

| | |
|---|---|
| | boolean::reconfigureNetwork(String deviceID, NetworkConfiguration networkConfiguration) |
| | void::reconfigureComponent(Component component, ConfigurationInstruction configurationInstruction) |
| | void:: UIRetrieved(String ComponentID, UI ui) |
| | void:: setAdaptedUI(UI ui, String componentID) |
| | void::retrieveState (String componentID) |
| | void::setState (String componentID, State state) |

Further details on these methods can be found in appendixes A and B.

## INTERACTIONS

Figure 12 illustrates the migration of a component from one device to another. While the specific methods are detailed in the appendixes, the value of this sequence diagram lies in the order of the events that occur in it.

After an initial registration phase, the source Open Client (OC) requests the available devices where it could migrate its components, based on the required capabilities.

Be it through a manual selection or a trigger from the Trigger Management, the OC decides to migrate one of its components, and does so by calling triggerMigration on the Open Server.

At this point, the component is launched in the target device, and paused in the source. The state from the source components is then extracted, and injected into the target components.

Upon completing this step, the source component is terminated, and the target one is started.

After some final cleanup, the migration of the components has concluded.

For further examples on the migration process, please refer to Section 8 which exemplifies sample migrations using the applications being developed in Open's WP5.

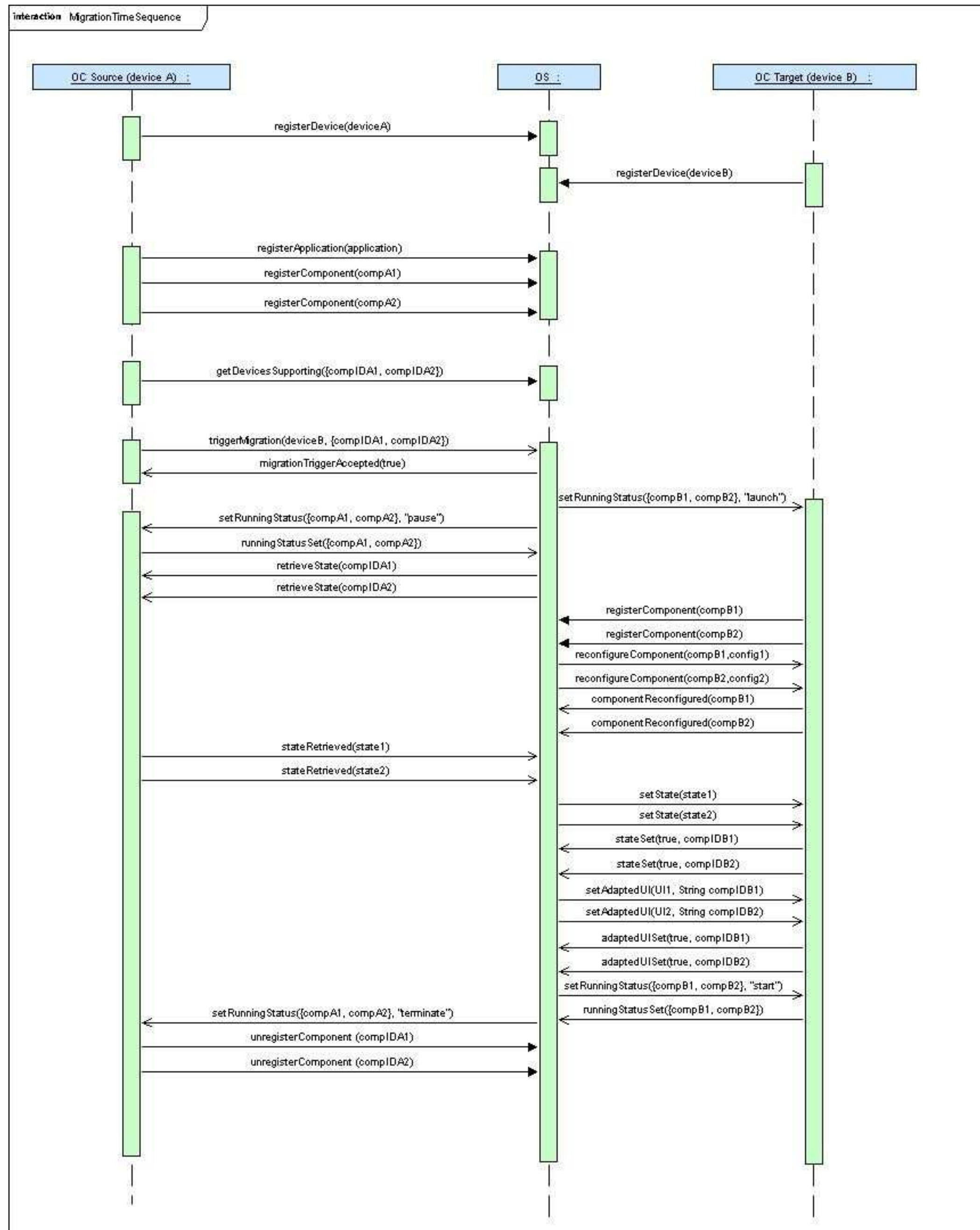


**Figure 12 Message sequence diagram for the migration of an application with two components**

## 7.1.3. STATE HANDLER

The State Handler component is responsible for handling the state information from applications. It does not deal with the state information internals, which it treats as a data “blob”, but rather concerns itself with transporting it between applications which are in the process of migrating.

Depending on the specific type of applications, the State Handler can take on different forms. We would like to highlight the special case of Web pages, where state is extracted (on the client side) using an injected JavaScript piece of code within the application, aimed at annotating the application in order to capture the current state of the application at the time when a migration is triggered. Then, this state information is handled by the Web State Handler module (on the server side) which is responsible to map it onto the user interface provided on the target device, so that the interaction can continue in a seamless way (from the user's point of view) on this new device. The Web State Handler component saves information like cookies and session_ID, as well as all the data provided/modified by the user while visiting the page. Indeed, starting from a web address, an HTTP request is sent (through a URLConnection) and the requested page is then saved. Then, the HTTP response is analysed in order to identify the existing cookies and any possible *session_id* existing in the querystring. From this point onward any request sent to that application server will contain the saved information.

The functionalities that this module provides are basically two:

- The first one is getting the user interface of the application to be migrated. This functionality can specify the application components to be migrated
- The second functionality is mapping onto/adapting for the target device UI the information that the user has included in the version of the page which was last visualized before migrating.

An example of how the Web State Handler is involved within a migration of a web application is described in Section 8.1

## OPEN INTERFACES

The Web State Handler module is only accessed by the Orchestrator on the Open Server, which needs to support the appropriate handler for each application type (e.g. web, desktop, etc…). It therefore does not have any Open interface methods, or rather, it mirrors those offered by the Orchestrator in what relates to state.

## INTERNAL INTERFACES

The first interface that Web State Handler module offers to the Open Server Orchestrator is the getUI method. This allows for getting the UI for a specified set of components that are considered for migration. A description of the method follows:

| **UI::getUI (String[] componentID, String callback)** | |
|---|---|
| componentID[] | The components to migrate |
| callback | Optional parameter. If it is missing then the request will be synchronous |
| *Returns:* | The user interface for the specified components |
| This method is called by the OS Orchestrator Request to get the UI for the specified components ( the callback parameter is optional, if it is missing then the request will be done synchronously). | |

The second interface that Web State Handler module offers to the Open Server Orchestrator is the setStateByServer method. This is aimed at adapting the state information with the interactive

application, taking into account the characteristics of the new device. A description of the method follows:

| UI::setStatebyServer(State[] states, UI logicalUIDescr) | |
|---|---|
| states | The state of the various components, that has to be mapped. It basically contains the mappings between state and target components. |
| logicalUIDescr | The logical user interface description (at the concrete level) |
| *Returns:* | A logical user interface description (at the concrete abstraction level) for the target device, updated through the state information. |
| This method allows for updating the concrete UI description for the target device through the state information. | |

### 7.1.4. TRIGGER MANAGEMENT

The Trigger Management component is responsible for prompting the migration orchestration component to initiate a migration. In a sense, this module behaves as a classifier that chooses when the states of the devices, the networks and the application should be changed in order to start a migration. It also suggests to which state the individual configurations should be changed.

Such a classifier takes its decisions based on the application's semantically described characteristics, the situation of the user, and the situation of the devices involved in the migration and properties of the underlying network. In most circumstances, said information can be derived from context retrievers, using the context management framework, and from the knowledge derived from the lower network layers using the performance monitoring component – potentially also through the context management framework.

A detailed illustration of the components relevant for the trigger management is seen in Figure 13.

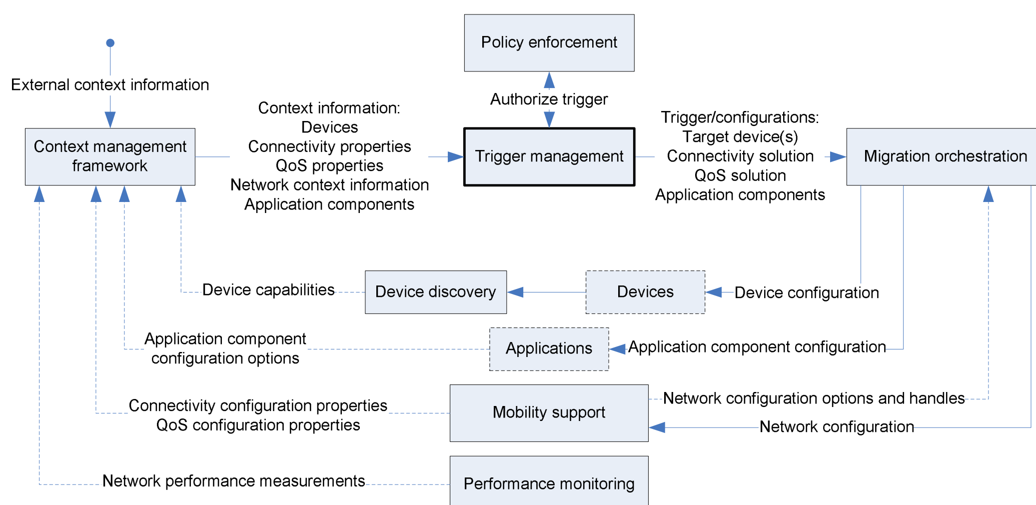


Figure 13: General interface specification for the trigger management module

## OPEN INTERFACES

The Trigger Management is an essentially active component. By monitoring the current situation, it may decide to call the triggerMigration method on the Orchestration component, thus initiating a migration. It therefore does not offer any interface, but behaves purely as a client.

In further detail, the situation monitoring will be based on collecting:

- **Registration of application configurations:** Takes a list of application component identifiers coupled with technological requirements to the run-time platform (such as javascript, flash, 3D-library, …), and QoS requirements to device and network in case of communication with other peers

- **Registration of network configurations**
  - **Connectivity configurations:** Takes a list of connectivity option identifiers characterized by relevant performance and reliability parameters
  - **Performance measurements:** Takes as input relevant dynamic network properties from current network topology and traffic
  - **QoS configurations:** Takes as input a list of QoS option identifiers characterized by relevant performance and reliability parameters

## INTERACTIONS

Figure 14 shows an example of an automatic migration trigger interacting with migration orchestration functions and mobility support functions. The Open Client (OC) at the source, is receiving data from an Application Server. As the network conditions change, an automatic migration trigger is generated that points the application server to feed the data (e.g. a video stream) to a different component, potentially in a different device.

In a real world situation, this example could represent a scenario where a user watches a video on his mobile phone. As QoS drops due to lack of coverage as the user moves, the system might trigger the migration, making the video available on the car built-in TV, which has a better network link at the moment.

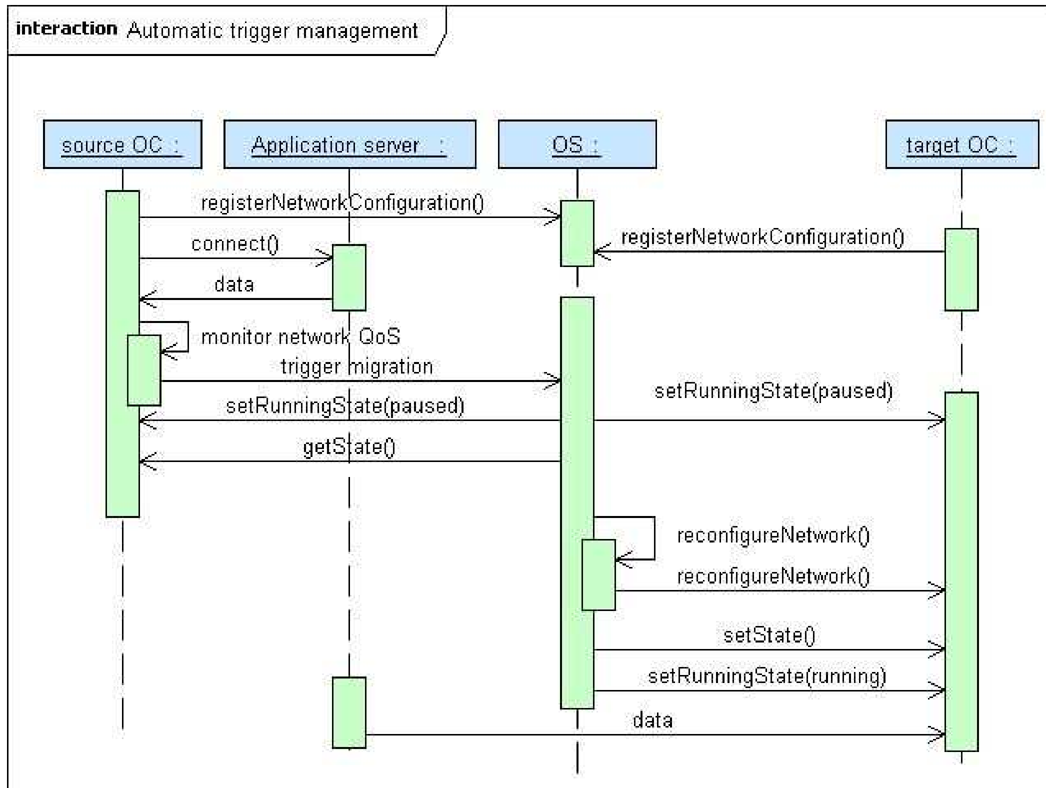


Figure 14 Message Sequence Diagram showing the automatic trigger for the migration of a server side data source

### 7.1.5. POLICY ENFORCEMENT

The policy enforcement component verifies if a migration proposed by the Trigger Management is possible according to a set of rules. The rules may be matched against the current situation, not unlike in the case of the Trigger Management. The Policy Enforcement module, however, does not make recommendations on migrations, but rather allows or forbids them.

#### OPEN INTERFACES

The Policy Enforcement module is only accessed by the Orchestrator on the Open Server. As such, it is not a public component, and therefore does not implement any of the Open Interface methods.

#### INTERNAL INTERFACES

The Policy Enforcement module offers an interface to the Open Server Orchestrator: allowMigration. This provides the PDP (Policy Decision Point) functionality that the module is responsible for. A description of the method follows:

| bool::allowMigration (String[] componentID) | |
|---|---|
| componentID[] | The components to migrate |
| targetDeviceID | The migration target |
| *Returns:* | A Boolean determining if the migration is allowed or not |
| This method is called by the Trigger Management component before sending a migration trigger to the Migration Orchestrator. It returns true, if it is possible to perform the migration, false if it is not possible (for example for some privacy requirements). | |

#### INTERACTIONS

This component interacts with the Context Management Framework, in order to retrieve the needed data, and responds to the Open Server Orchestrator on migration decisions.

### 7.1.6. MOBILITY SUPPORT (SERVER SIDE)

The primary goal of the mobility support function is to make mobility transparent to other functions. These functions can be either within the OPEN platform or in the application. The mobility supports function has to hide any mobility during migration and still allow for continuity.

Several mobility methods were introduced in (2): NAT, Mobile IP, SCTP and SIP. Deciding the method used in the mobility process is not a trivial decision because involves the current situation at the source device. Which applications are currently running? Which one should I migrate? Will there be consequences for the rest if I change the IP? Etc.

There is no optimal solution applicable in all the cases meaning it will be a task of the mobility support function to evaluate the scenario conditions at each moment and decide which is going to be the most suitable method to perform the mobility with absolute transparency.

For example, if the user is running only one application and wants to migrate it, mobility support could choose NAT as a method since the change of IP in the source device will not affect other applications because they do not exist. On the contrary, if the user is using different applications and only wants to migrate one of them, methods affecting IP addresses would not be a good solution because the connectivity in the other applications will be affected.

Another task that belongs to the mobility support function is updating the CMF with relevant information related to available networks for each client and its characteristics/performance. This task is introduced in the client side components.

Figure 15 depicts in detail all the mobile support interactions within OPEN Platform.

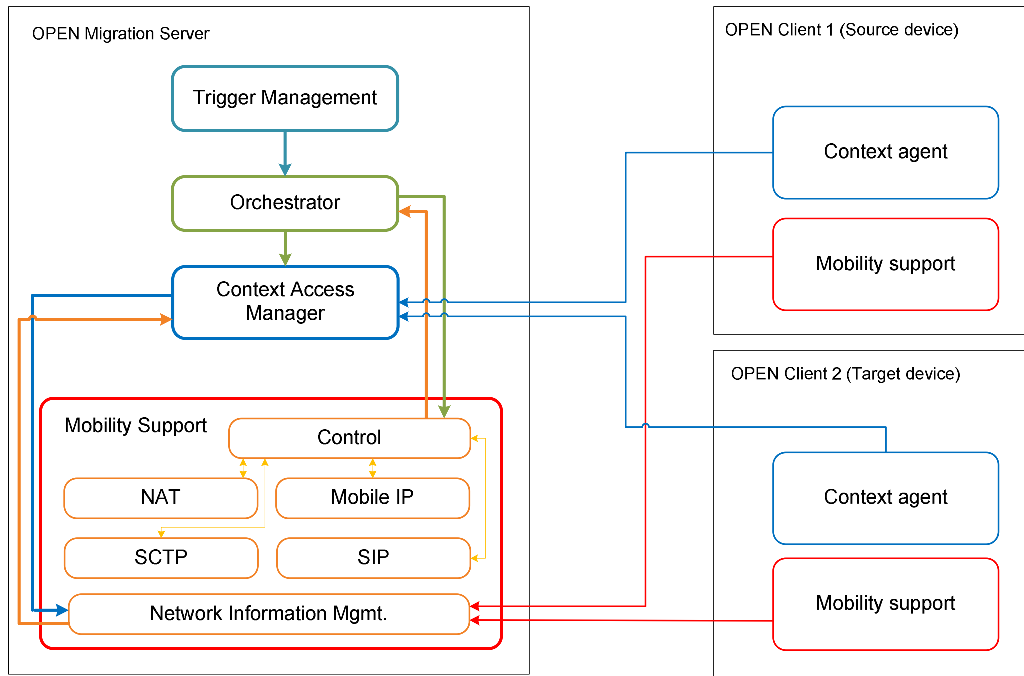


**Figure 15 Mobility support interactions within the OPEN Platform**

## OPEN INTERFACES

The Mobility Support component may call on its correspondent adaptor, or on the application to reconfigure a certain network connection using the following method:

| Open Client methods at the Mobility Support component | boolean::reconfigureNetwork(String deviceID, NetworkConfiguration networkConfiguration) |
|---|---|

Likewise, client side components will keep the Mobility Support component updated on network availability by registering their network connections and reporting on the effect of changes triggered by the server, using the following methods:

| Open Server methods at the Mobility Support component | void::networkReconfigured(String deviceID, boolean isReconfigured) |
|---|---|
| | boolean::registerNetworkConfiguration(String deviceID) |

As usual, the details of these methods can be found in appendixes A and B.

## INTERACTIONS

In its basic functionality, the Mobility Support component will trigger the reconfiguration of networks in straightforward migrations. It will instruct both ends of the migration on what connections to shut down, and which ones to establish.

As development progresses, we envision the mobility support not only as a passive function but also as information provider. Specifically, it will provide information related to the networks availability and performance to Context Access Manager. This information will be sent either periodically/event based or after receiving a query from the Context Access Manager. Next, Figure 16 to Figure 18 shows an example scenario and Figure 19 shows its sequence diagram respectively.

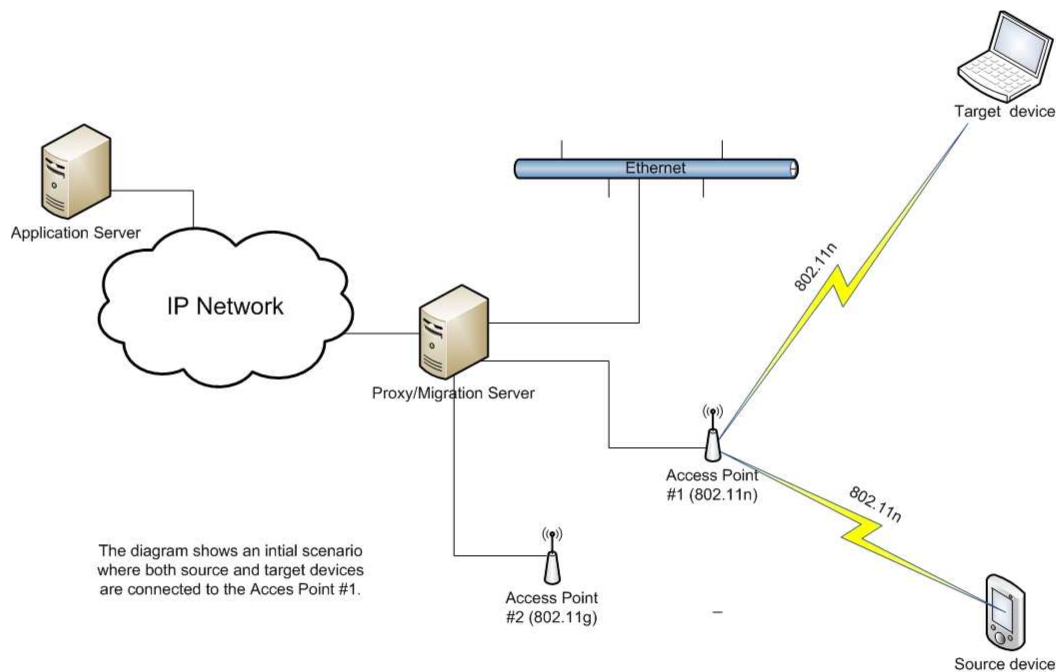


Figure 16 Scenario A description

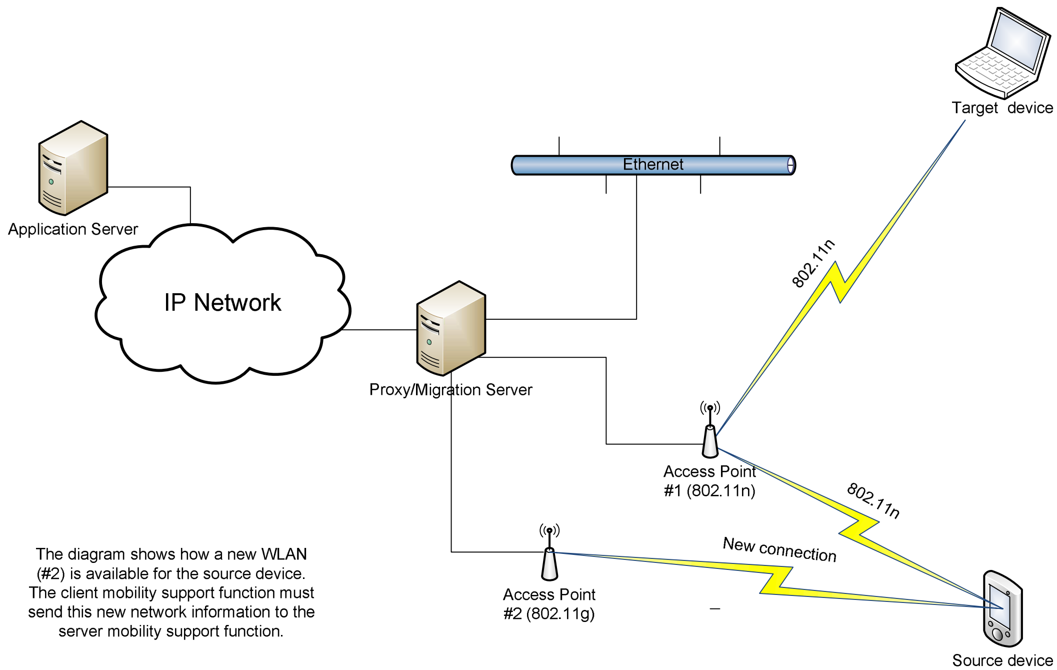


**Figure 17 Scenario B description**

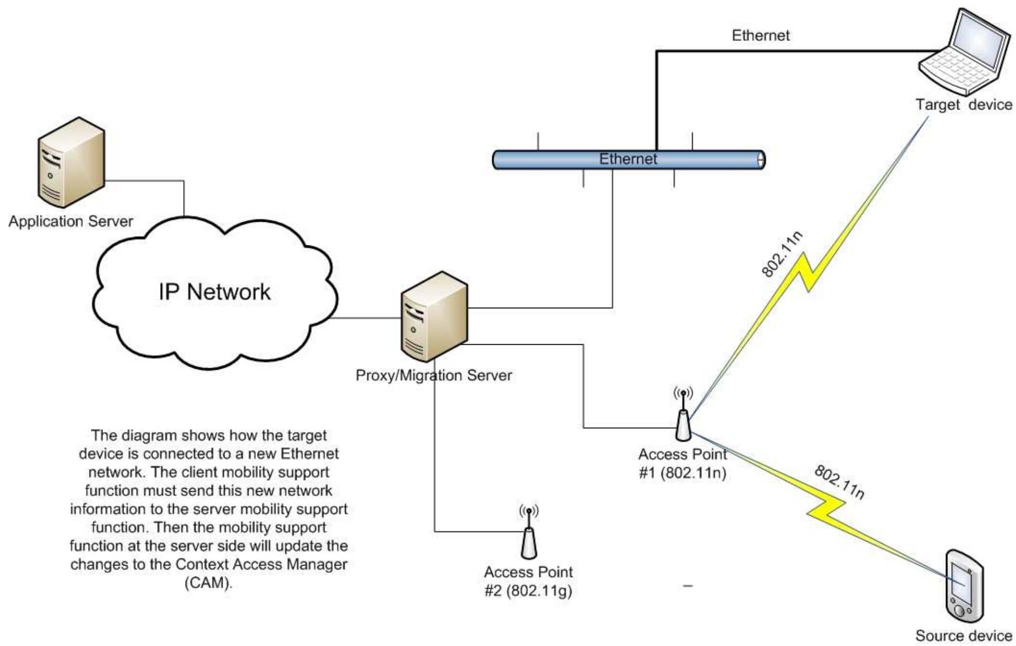


**Figure 18 Scenario C description**

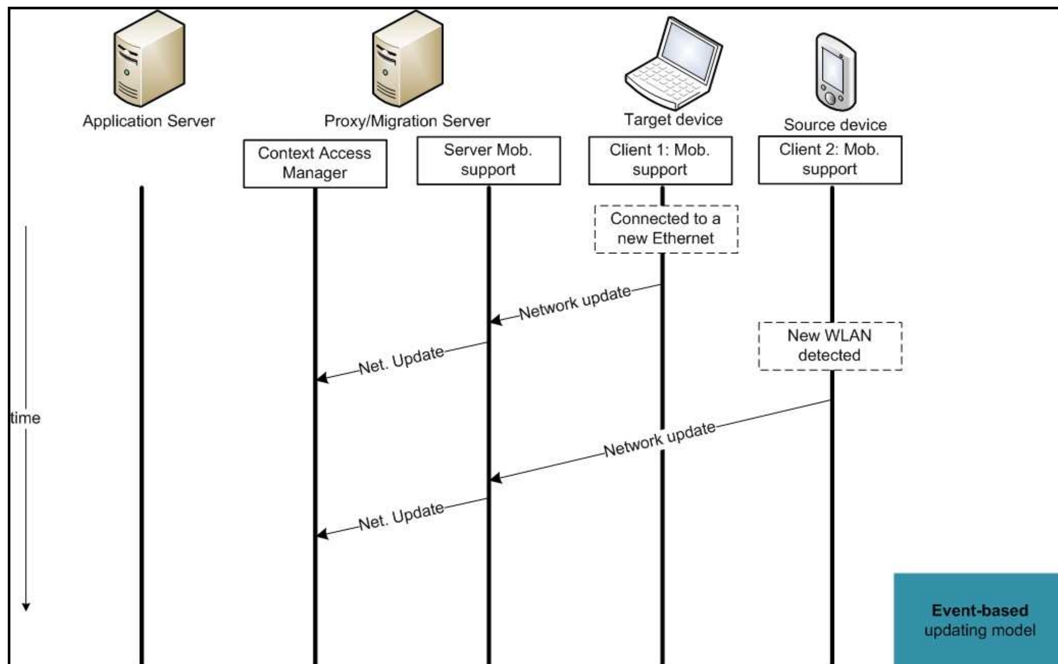


Figure 19 Network Information Provision sequence diagram. (Event-based model)

### 7.1.7. UI ADAPTATION (WEB)

UI Adaptation, on the Open server side, refers to the adaptation of Web User Interfaces, and is complementary to the adaptation of Java interfaces presented in Section 7.2.3.

This module, implemented in Java, generates a logical description of the UI rendered on the source device and transforms this description into a new UI adapted for the target device. This module exposes the following functionalities: one is aimed at building a Concrete User Interface (CUI); the other one gets the CUI for the source device and generates an adapted CUI for the target device. Another functionality/interface that is provided by this component is that of generating the final UI for the concerned platform.

#### OPEN INTERFACES

This component provides Web specific solutions for the retrieval of User Interface and state information from web pages. This functionality, however, is exposed by the Orchestrator, and therefore no Open interface methods are directly implemented.

#### INTERNAL INTERFACES

As explained in the previous section, the following methods are provided internally by the Web UI Adaptation module, mirroring those offered by the orchestrator, which are part of the Open interface (Appendix B contains more details on these interfaces)

| Open Server methods offered internally at the Web UI Adaptation component | void:: retrieveUI(String componentID) |
|---|---|
| | void:: adaptedUISet(Boolean isUpdated, String componentID) |
| | void::retrieveState (String componentID) |
| | void::setState (String componentID, State state) |

#### INTERACTIONS

This module does not call any method/function of other modules. Section 8.1, however, provides an interesting example of a full migration that involves the Web UI adaptation.

### 7.1.8. APPLICATION LOGIC RECONFIGURATION (ALR)

The server side Application logic reconfiguration module (ALR) supports applications by the dynamic adaptation of the application logic to their specific needs in constantly changing situations. At this, an application is divided into two parts, namely the reconfigurable application logic, and the rest of the application which could be among others static application logic and the User Interface. The ALR module is responsible for the adaptation of the reconfigurable part of the application logic. At this, we offer two types of adaptation like introduced in deliverable (5), namely the change of the wiring between components and the adaptation of their behavior. To perform these tasks, several interactions between the reconfigurable application logic on the client side and the ALR module on server side are required. For this, the application logic components have to implement the *OPEN Client Interface* directly, or by using a proxy or an adapter. On the other hand, the ALR module implements methods of the *OPEN Server Interface* to let the application interact with the platform. The ALR-related methods of both interfaces will be explained in the following.

The communication between the reconfigurable application logic part and the rest of the application can be done using an arbitrary protocol, like Web Services (PacMan prototype) or JSON over HTTP (Emergency Scenario prototype).

#### OPEN INTERFACES

The ALR may call on the following methods from an application or client side component:

| Open Client methods at the ALR component | void::setRunningStatus (String componentID, String runningStatus) |
|---|---|
| | void::reconfigureApplication (String applicationID, Configuration configuration) |
| | void::reconfigureComponent(Component component, ConfigurationInstruction configurationInstruction) |

And therefore, expects replies on the corresponding asynchronous interfaces:

The ALR module works on the following methods (further explained in appendixes A and B.)

| Open Server methods at the | void::runningStatusSet (String componentID, String runningStatus) |
|---|---|
| | void::ApplicationReconfigured(Boolean isReconfigured, String applicationID) |

| ALR component | void::componentReconfigured(Component component) |
|---|---|

In summary, if a component of the application logic has to be reconfigurable, it has to implement these three methods of the Open Client accordingly. If a component of the application logic is not able to implement these methods directly, a dispatcher can be used which takes the calls from the ALR module and forwards the call to the right component. Thus, the concept presented here enables the developer to implement the application logic in the preferred programming language, as long as the ALR method can call methods at the components, directly or by using a wrapper.

## INTERACTIONS

The diagram in Figure 20 shows a typical reconfiguration, where, once the components have been registered, the application decides to reconfigure one of the components by first stopping it, then reconfiguring it, and finally restarting it.

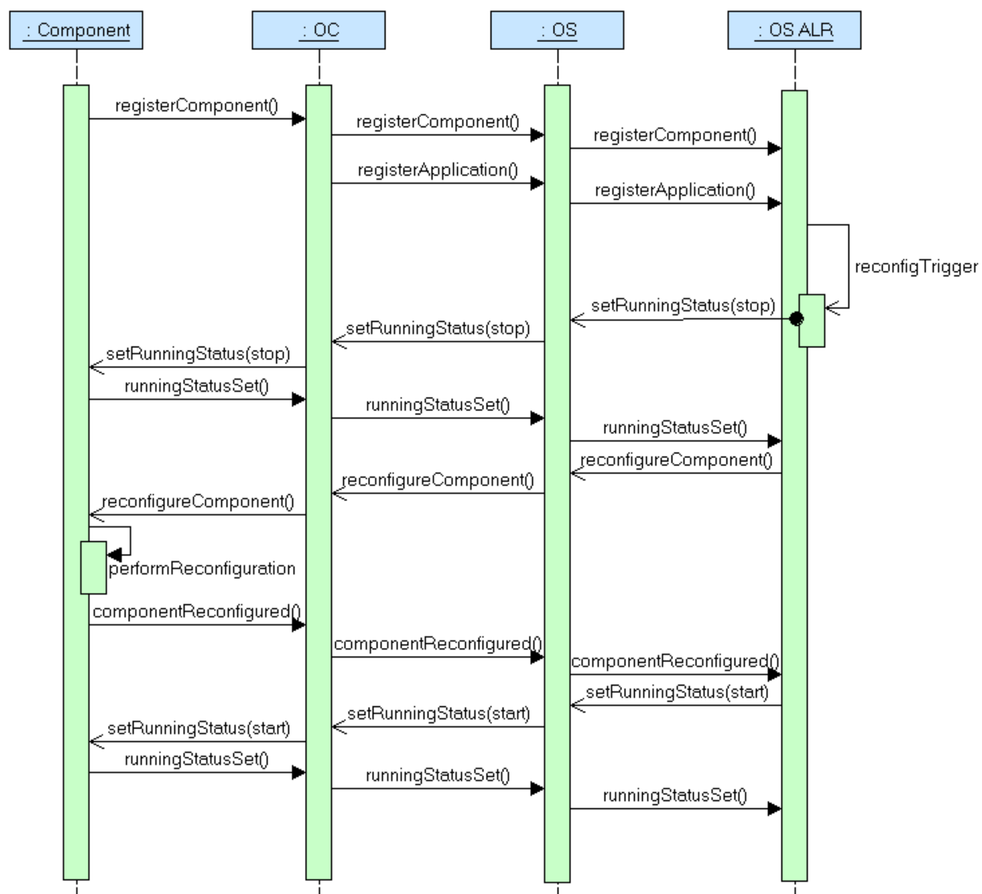


**Figure 20 Reconfiguration of a component's behavior initiated and controlled by the ALR module on server side. The reconfigTrigger can be a migration or change of context of one or more components.**

## 7.2. OPEN ADAPTORS IN THE OPEN CLIENT

### 7.2.1. CMF (CONTEXT AGENT)

The CMF on the client side is also a Context Agent, hence has same basic capabilities as the server side Context Agent. The difference, however, is the role of this Context Agent, which is only to retrieve local context, and to provide locators of this information to the CMN (Context Management Node, i.e. the server side Context Agent) for later lookup. Except for that, the interaction and interfaces to the agent on the client side as seen from an application or component view, is identical to what has previous explained in section 7.1.1.

### 7.2.2. MOBILITY SUPPORT (CLIENT SIDE)

Different kinds of mobility were introduced in (2): personal, referred to the ability to reach a mobile user through devices currently available to the user; terminal, when a device is reachable by a correspondent node while moving between networks; session, when user sessions can be maintained while moving between terminals; and finally service, when the device is changed without affecting the access to the same services. Service mobility combined with session mobility is called service migration.

In OPEN, mobility means that components of the application, providing the service to the user, moves during the service session. In general the moving components can be located both server-side and client-side in the communicating application, but in OPEN we only consider client-side components in the first year version. Thus, the mobility type to be supported in the platform is service mobility.

The mobility support function at the client side consists of a simple passive module that interacts with the mobility support function at the server side (Figure 21). The mobility support function main goal is to provide service mobility to the rest of functions in a transparent way.

Apart from providing transparent mobility, subsequent stages envision the client mobility support function as the way of providing all the information related to networks to the mobility support server side. This information will basically consist of the available networks for each device, characteristics and performance. Therefore, aspects like estimating the QoS of the applications in the target device can be managed.

Thus, summarizing, there will be interaction between all the mobility support modules at the client side and the mobility support function at the server side. This last one will group all the information provided for all the clients and will send it to the Context Access Manager using a retriever.

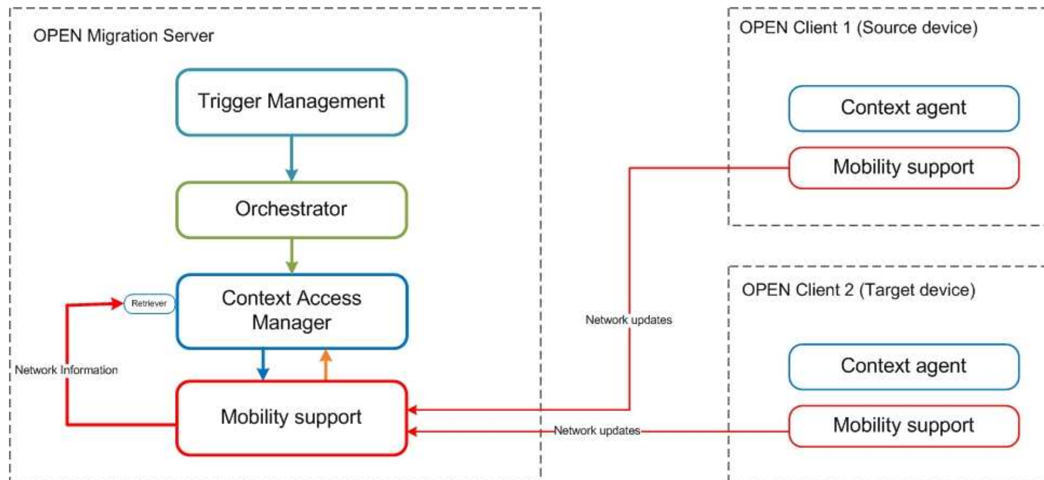


**Figure 21 Mobility support server-client side interaction**

The interfaces of the Mobility Support module match those in the server side, and implement the Open Client and Open Server interfaces where appropriate. Please refer back to section 7.1.6 for further details.

## INTERACTIONS

The mobility support function at the client side must interact with the mobility support function at the server side by sending network information updates. These updates will initially consist of a list of available networks and its characteristics: type of network, bandwidth, delay, jitter, congestion alarms and restricted ports. The list of restricted ports could be useful information since some application initially running in a network can be blocked by firewalls after migration into a new different network.

All these information will be forwarded to the Context Access Manager by using the correspondent retriever as shown in Figure 21.

## WEB APPLICATION LOGIC RECONFIGURATION

Web Application Logic reconfiguration has certain peculiarities that warrant further in-depth explanation.

A web application is able to dynamically update its logic, which is separated from the user interface. The Application Logic Reconfiguration (ALR) is responsible for keeping up to date with the logic of the application based on the current contextual state.

Application Logic components, implemented as web services, register to the ALR and allow the web application to get the updated logic configuration according to the current state. As an example, whenever a ghost of the Pacman game has to choose one among many possible directions (i.e.: it is over a crossing), the web application asks the ALR for the new direction of the

ghost. This is done by calling a specific web service and by passing to it all the information needed by the ALR to compute the new ghost direction. Information to be passed may include (but not be limited to) the position of the pacman and of all the ghosts.

| **newValue::getNewParameterValue(currentApplicationState)** | |
|---|---|
| currentApplicationState | the value of the set of parameters defining the application state. In practice, this would be a set of strings and/or integers to be passed as parameters to the web service, or even a single string encoding the whole application state. |
| *Returns:* | the new value of the requested parameter computed by the ALR according to the input application state. |
| Creates the GUI components according to the layout specified in the XML file that contains the concrete GUI description for the respective device. | |

### 7.2.3. UI ADAPTATION FOR DESKTOP APPLICATIONS

By using the Open Client Adaptors, Desktop (i.e., non-web) applications can adapt their user interface to the enhanced possibilities of Multicore devices. This type of adaptation is disjoint from the one performed for Web pages, presented in Section 7.1.7.

The UI Adaptation adaptor of the OPEN platform handles the task of generating an adapted GUI of a Java application according to the capabilities and characteristics of the target device. In order to use this module, the application programmer has to specify an XML description of the GUI layout of the application. During the application migration process this XML description is fed to a sub-system of the Web (non-Generic) UI Adaptation component (see section 2.2.6), which will transform it to an adapted XML description that takes the characteristics of the target device into account. The migrated application on the target device can then use this adapted description to re-build the GUI.

Instead of hard-coding the creation of GUI widgets in the Java source code, the developer has to use a Java library that is provided by the Generic UI Adaptation module: This package parses the XML GUI description and creates the corresponding GUI widgets on the fly (e.g. at application start-up time). Furthermore it will hand back a (possibly hierarchical) data structure that allows the programmer to access the created GUI components from the application logic code.

#### LOW-LEVEL INTERFACES

The Desktop UI adaptation operates through the use of Java libraries linked to lower-level, C applications that exploit the multicore capabilities. Java applications can use the following method to adapt their interface:

| **GUIInfo::createGUI(ConcreteGUIDescXML GUIInfo)** | |
|---|---|
| ConcreteGUIDescXML | An XML file that contains the concrete GUI description for the respective |

| | device. |
|---|---|
| GUIInfo | This data structure maps identifiers (name strings or IDs) to references to the created GUI components (like buttons, labels, etc.). This allows the application logic code to access the created GUI widgets. |
| *Returns:* | Void |
| Creates the GUI components according to the layout specified in the XML file that contains the concrete GUI description for the respective device. | |

### 7.2.4. OPEN CLIENT DAEMON WITH UI

This module represents the Open Client running on the user mobile device. It includes the Device Selection Map, formerly Discovery Map, which supports the discovery of available devices

The Open Client Daemon is written in C# and deployed on the user device, while the Context Agent (also deployed on the user's device) is defined by a set of Java classes. Thus, for integrating the Open Client and the SCMF (i.e.: to enable interaction between the two modules), a lightweight Java Virtual Machine suitable for PDAs (Mysaifu JVM) has been exploited.

The strategy that seems, so far, the most efficient is explained in the following.

The Context Agent is started whenever the Open Client is launched. It is the Open Client that creates an instance of JVM for executing the main class of the Context Agent (i.e.: the **ContextAgent.class** of the **de.nec.nle.contextagent** package).

In detail, the C# code for launching the JVM is the following:

```
Process contextAgentProcess;
string ca_logFileName = "ca_log.txt";
string jvmExecutable = "\\Programs\\Mysaifu JVM\\jre\\bin\\jvm.exe";
string jvmPar = " -cp \"\\SCMF\\lib.jar;\\SCMF\\external.jar;\\SCMF\\;\"
-Xmx10M -Xlogfile:" + ca_logFileName +
" de.nec.nle.contextagent.ContextAgent \\SCMF\\CMN.xml";
contextAgentProcess = System.Diagnostics.Process.Start(jvmExecutable, jvmPar);
```

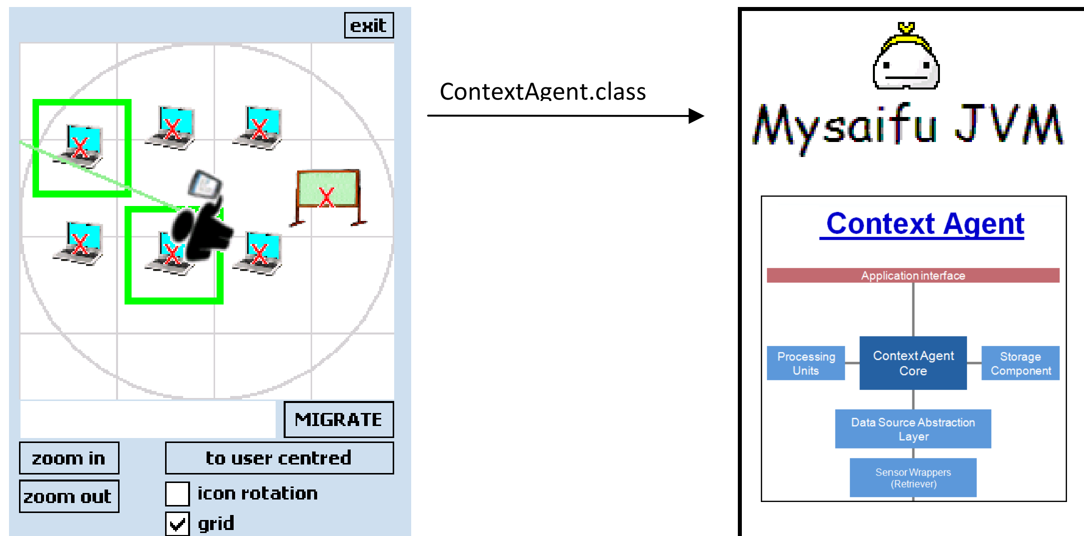


Figure 22 Client Daemon interaction with the CMF

By holding the `contextAgentProcess` reference it is possible to close the JVM from C# environment. The Context Agent might not be running for an unlimited time. Thus, on closing the Open Client, `contextAgentProcess.Kill()` is executed.

The Migration Client also opens and continuously shares the output stream of the JVM for getting the state of the Context Agent (i.e.: to know whether it is running). This is done by opening the log file of the JVM console in "read only" way:

```
FileStream fs = File.Open( ca_logFileName, FileMode.Open,
FileAccess.Read, FileShare.ReadWrite);
```

## OPEN INTERFACES

The Open Client Daemon implements parts of the Open Client Orchestrator functionality to handle migrations. This is specially the case for web applications: using the Open platform, scripts are injected into the page to retrieve state, and indeed, the UI is adapted to the target device. The interface, however, is never enriched with migration controls to preserve the page contents, and therefore a separate UI is needed. Additionally, the Open Client Daemon can be used for device selection, and to trigger manual migrations.

The Open interfaces provided by this adaptor are, therefore, those of the Orchestrator, as explained in Section 7.1.2.

## INTERACTIONS

The following diagram (Figure 23) shows how the Open Client Daemon is used to trigger a migration of a component.

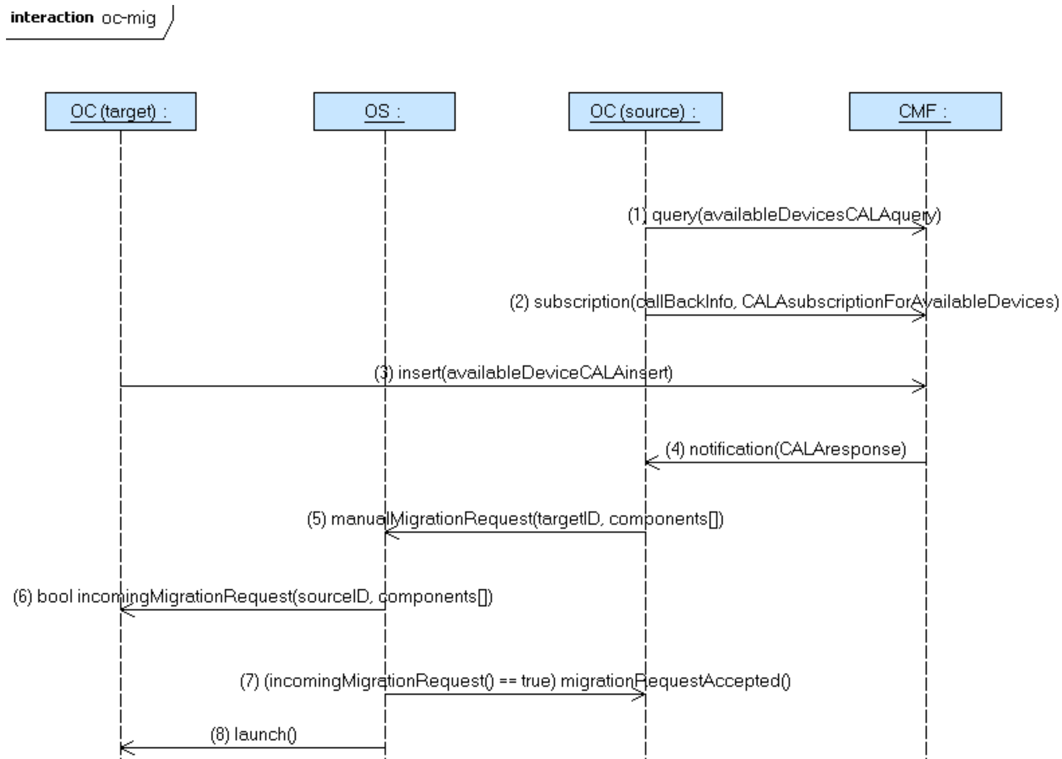


**Figure 23 Sequence diagram of the manual migration trigger with respect to the Open Client Daemon.**

The steps depicted therein are as follows:

1. The Open Client Daemon (in the following: OC) of the source device, after being started, sends a CALA query to the CMF asking for the list of currently available devices (i.e.: candidate target devices for migration)
2. The source OC subscribes for new available devices (in order to be notified whenever a potential target device joins the smart environment)
3. A new target device joins the smart environment: the target OC sends an insert to the CMF specifying its parameters (that are embedded in the CALA string and that depend on the entity format)
4. As soon as the new target device has joint, the CMF notifies the subscriber (source OC).
5. The user of the source OC (who is interested in migrating one or more component towards the new target device) selects the target device in the OC interface and requests migration. The manual trigger is initially sent to the OS (since the source OC should not know the IP address of the target OC)
6. The OS, which is aware of the actual address of the target OC, sends an incoming migration request message to the target OC

7. When the target OC accepts the incoming migration request, a feedback message is sent to the source OC

8. A launch message is sent to the target OC in order to start the adapted (parts of) migrated application

Note that, even if the CMF is shown in the sequence diagram as a separate entity, it is actually a distributed system. The CMF lies in the Open Server (as Context Management Node – CMN) as well as in each OC (as Context Agent – CA). Thus, whenever a OC needs to forward or to retrieve data, it just “contacts” the local interface (CA) that manages the synchronization with the server (CMN).

### 7.2.5. MIGRATION ORCHESTRATOR CLIENT

This component manages applications migrations on the OPEN client side. In our prototypes, it will be merged with the Open Client Daemon, which offers the same interfaces. Further details can be found in Section 7.1.2

### 7.2.6. WEB STATE HANDLER (CLIENT SIDE)

This module (which is a JavaScript component) manages the application state on the OPEN client side. Basically, it has to check if a manual migration has been triggered and, whenever a migration request is received, it captures the state of the page and sends this state to the Orchestrator. Then, the Web State Handler (server side) will manipulate such information in order to deliver an adapted state that will be used for providing the new application that will be uploaded on the target device.

## 7.3. DISPATCHERS

Dispatchers have already been introduced in Section 4. This section provides further insight in how components may use them to communicate.

Notice that, because of the choices in interface design, there is only one dispatcher implementation, which can be deployed in multiple points of the platform, as shown in Figure 7 (page 11). Said dispatcher will always present an Open Client interface to the Client side, and a Server one to the server side.

### OPEN INTERFACES

Dispatchers are unique in that they expose the complete Open Interface. Far from complexity reasons, this is due to their proxy nature: every component in the Open platform will talk to a Dispatcher, which will handle the proper routing of the messages, regardless of whether that is:

- Towards the right component that it proxies for
- Between the Open Server and an Open Client

In both cases, it will rely on the Mobility Support module to detect changes in network infrastructure.

### INTERNAL INTERFACES

Beyond the Open interfaces, the dispatcher components need to offer the methods required to register and de-register components, so it can keep track of where incoming messages should be routed. To this effect, the following interfaces are used:

| **String::registerListeningOpenComponent(String openComponent, URL endpoint)** | |
|---|---|
| openComponent | The type of component that is being registered (e.g. orchestrator, ALR) |
| endpoint | The URL of the XML-RPC endpoint where messages should be sent |
| *Returns:* | A String representing the registrationID, which can be used to get pending requests or unregister |
| This method is called by an Open Client or Server component, as well as by applications, to register their functionality at the appropriate Dispatcher. In doing so, the Dispatcher will bind the methods associated to the component to the provided endpoint. For instance, once this call is done, all incoming “triggerMigration” calls will be forwarded to the endpoint registered for the “Orchestrator” component | |

| **String::registerPollingOpenComponent(String openComponent)** | |
|---|---|
| openComponent | The type of component that is being registered (e.g. orchestrator, ALR) |
| *Returns:* | A String representing the registrationID, which can be used to get pending requests or unregister |

| This method is called by an Open Client or Server component, as well as by applications, to register their functionality at the appropriate Dispatcher. In doing so, the Dispatcher will know that a component is available to satisfy the given methods, and that it will poll the dispatcher to get pending requests at regular intervals. |
|---|

| **void::unregisterOpenComponent(String registrationID)** | |
|---|---|
| registrationID | The registrationID returned by the either of the registration methods |
| *Returns:* | Void |
| This method is called by an Open Client or Server component, as well as by applications, to announce to the dispatcher that they do no longer support the methods associated to the component type. | |

Additionally, when polling mode is used, components will need to poll the dispatcher interface to obtain the buffered messages using the following method:

| **String[]::getPendingRequests(String registrationID)** | |
|---|---|
| registrationID | The registrationID returned by the either of the registration methods |
| *Returns:* | An array of Strings representing the XML-RPC requests that were buffered and pending retrieval through the polling mechanism |
| This method is called by an Open Client or Server component, as well as by applications, to retrieve the XML-RPC requests that have been buffered by the dispatcher. | |

Finally, dispatchers at the edge of the Open Client that wish to be listen for Open Server calls (those not in polling mode), need to register to the dispatchers at the edge of the Open Server. To do so, the following methods are used:

| **String deviceID:: registerOpenClientDispatcher(String serviceEndPoint)** | |
|---|---|
| serviceEndPoint | the URL where this Open Client Dispatcher is listening |
| *Returns:* | A String with the deviceID that's assigned to this Open Client at registration time |
| This method is called by the dispatcher at the edge of the Open Client to register the Open Client to the Open Server. In return, it receives a deviceID which can be used by applications in the future. | |

| **Boolean success:: unregisterOpenClientDispatcher(String deviceID)** | |
|---|---|
| deviceID | The deviceID that we want to unregister |
| *Returns:* | True if the unregistration was successful, false otherwise |
| This method is called by the dispatcher at the edge of the Open Client to detach itself from the Open Platform. After this call, the Open Server will no longer consider this device to be an Open Client under its control. | |

Finally, applications need to know the ID of the device they are running inside of, in order to register themselves with the Open Server. For this purpose, the following method is offered:

| **String deviceID:: getDeviceID()** |
|---|

| *Returns:* | The deviceID that was assigned to this Open Client by the Open Server |
| --- | --- |
| This method is called by a component that requires the deviceID (usually for registration at the Open Server) | |

### INTERACTIONS

Figure 24 shows the operation of the polling mode. A server side component sends an XML-RPC request to the dispatcher, which buffers it, instead of sending it forward.

When, after a poll interval timeout, the Client component wishes to retrieve pending methods, it calls getPendingRequests on its closest dispatcher. This will then ask the next dispatcher in the chain, until it reaches the Open Server. Eventually, the pending requests at all the dispatchers are collected and delivered synchronously to the client.

After processing them, the client side can respond asynchronously using the methods designated for this purpose on the Open Server interface.

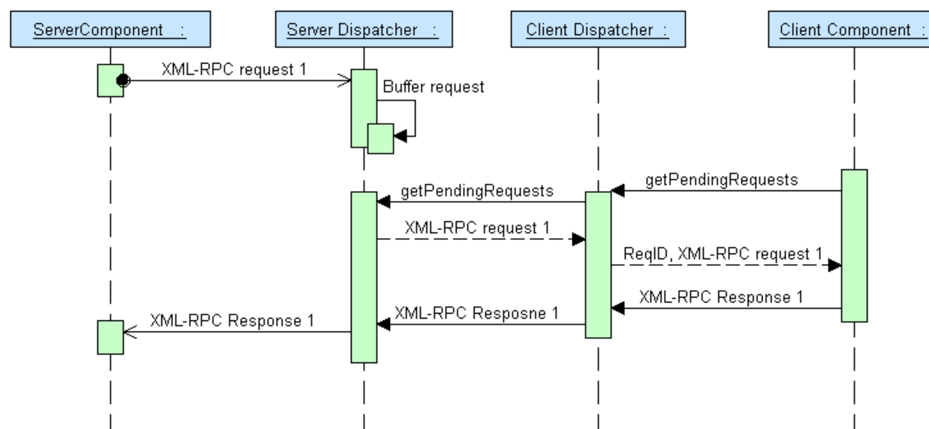


**Figure 24 Example operation of the polling mode in the dispatcher**

## 7.4. COMPONENT DEPENDENCY

The Open components and example applications are closely related, with modules depending on other modules to perform their functionalities.

In order to coordinate our work, we have identified the dependency graph shown in Figure 25. In it, components depend on the components they point to. At the top, we have applications which require the whole platform for migration, and at the very bottom is the UI adaptation and the Context Management Framework, which perform the lowest-level operations.

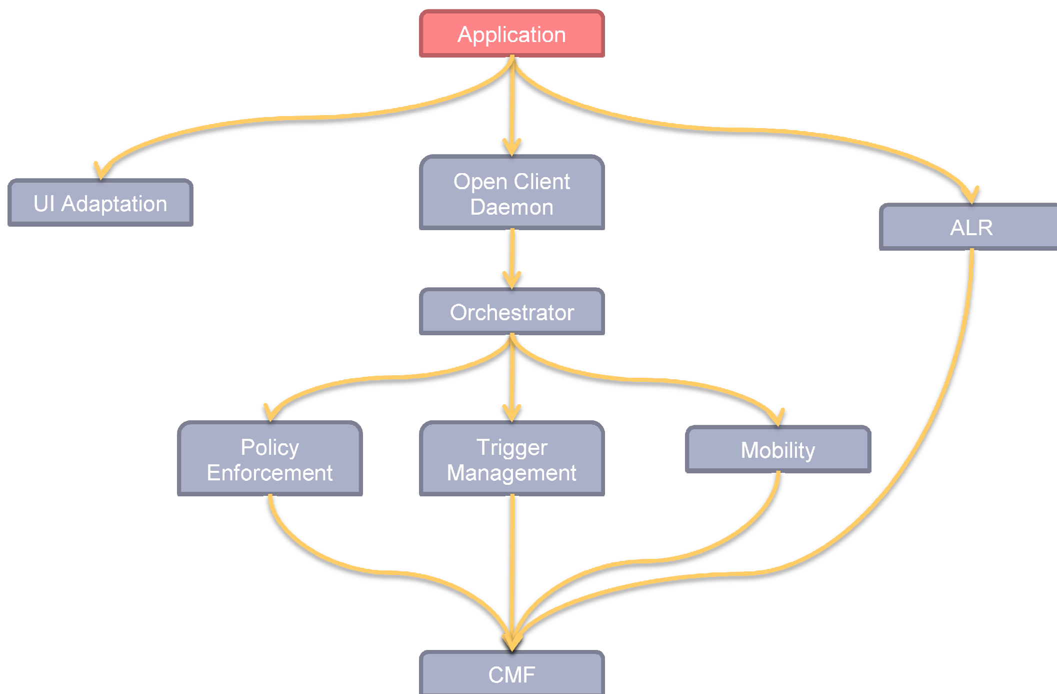


**Figure 25 Dependencies among the project's implementations. Upper components depend on the lower components**

A good understanding of this graph is being used to coordinate and prioritize the Open work.

# 8. OPEN PLATFORM OPERATION EXAMPLES

This section aims at providing an overview of the platform functionality, by showing how two sample applications integrate with the Open Platform components in order to migrate and adapt their components.

## 8.1. MIGRATION OF A WEB APPLICATION

Figure 26 shows the migration of a Web Application. In it, the application (source device) asks the OS Orchestrator for retrieving the user interface (retrieveUI) of the specified component. Then, the OS Orchestrator asks the OS Web State Handler to get the user interface for the specified components, also specifying an optional callback (if it is missing then the request will be synchronous).

Afterwards, there is the callback method UIRetrieved() from the OS Orchestrator to the application (source device). Then, there is the possibility either of a manual migration request from the OC Orchestrator (source device) to the OS Orchestrator (manualMigrationRequest() function), or an automatic migration trigger coming from the OS Trigger Manager (startMigration() method).

After receiving one of such requests, the OS Orchestrator asks the OS Policy for the authorization for allowing a migration and, if the answer from the OS Policy is positive, then the OS Orchestrator asks the OC Orchestrator (source device) to retrieve the state of a specified component. After the state has been retrieved, the OS Orchestrator can pause the application on the source device and then it asks the OS Web Adaptation to adapt the user interface (which means reverse engineering the retrieved web page, getting a logical UI description of it, and semantically redesign it for the target platform).

Then, the OS Orchestrator asks the OS Web State Handler to set the state of the user interface, then generates the new user interface for the target device (namely, produce a user interface in a specific implementation language). Then, the OS Orchestrator asks the OC Orchestrator (target device) the permission to activate a migration on the target device. If the answer is positive, the application on the source device has to be closed, and then the adapted UI is uploaded to the OC Orchestrator on the target device (setAdaptedUI()). After receiving the callback (adaptedUISet()), the OS Orchestrator can start the application onto the target device.

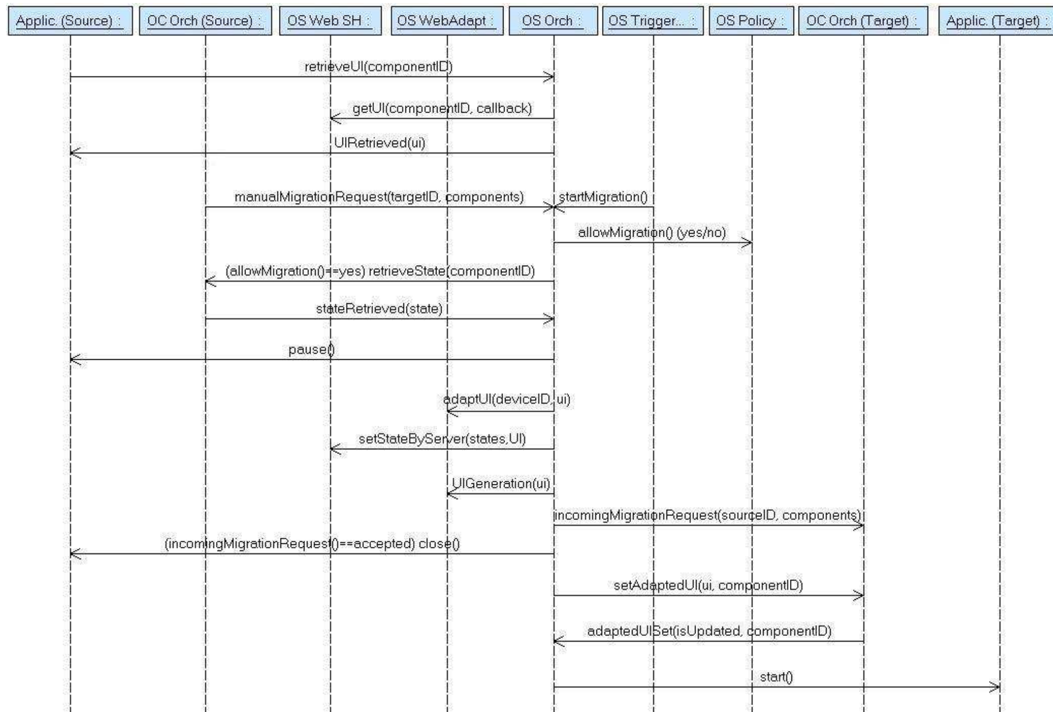


**Figure 26 Message Sequence Diagram for the migration of a web application**

## 8.2. COMPONENT MIGRATION IN THE SOCIAL GAME

The diagram describes the migration of a component of the Social Game from a source device to a target device, keeping the application running on the source device. Note that the application and the remaining components on the source device need to be reconfigured in order to work properly without the migrated component. To this end, the methods *reconfigureComponent* and *reconfigureApplication* with proper configuration information have to be called both on the source device and the target device.

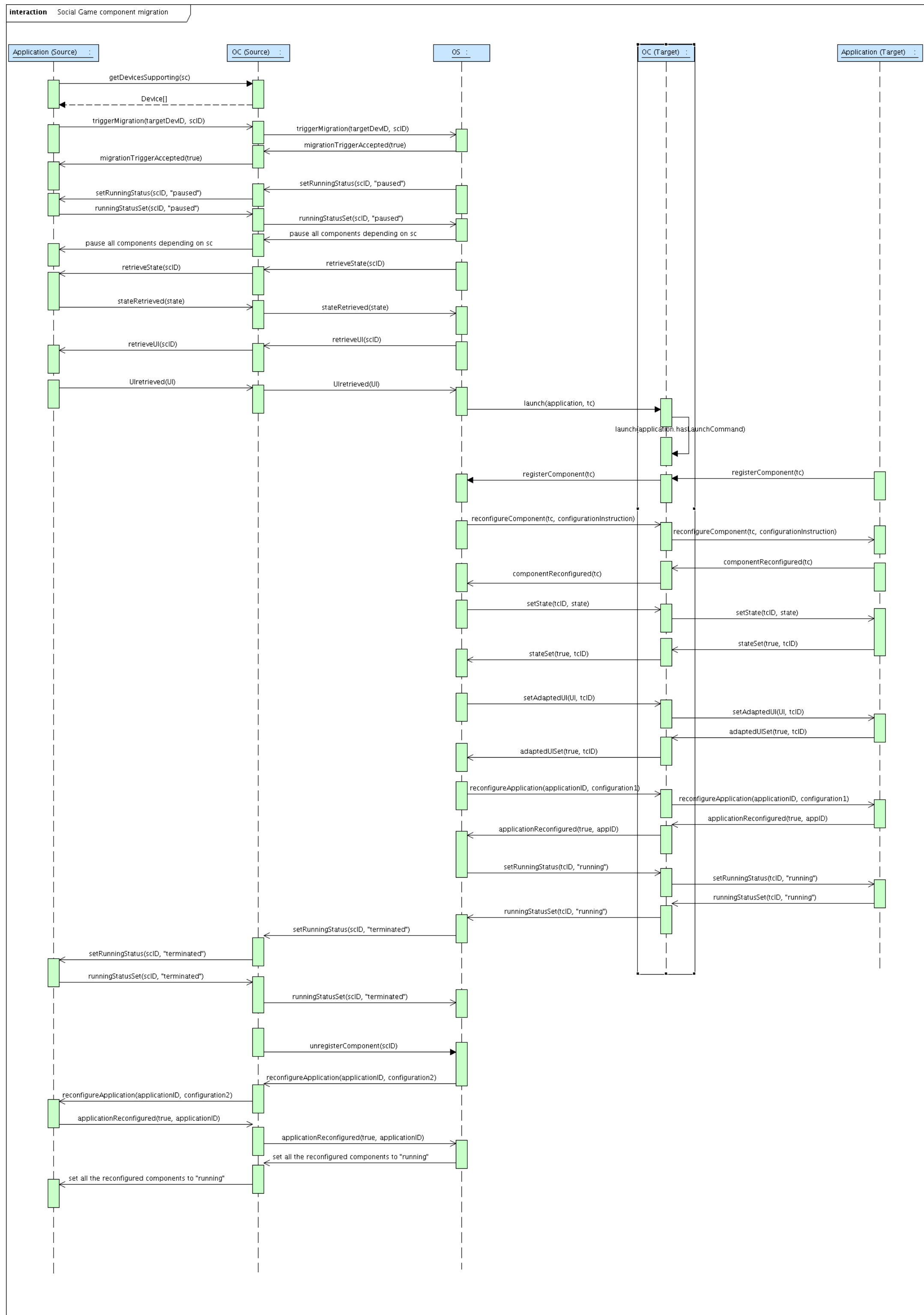


**Figure 27 Message Sequence Diagram for the Social Game component's migration**

# 9. CONCLUSIONS

Since the beginning of the work package's activities, WP4 has strived to harmonize the contributions in the rest of work areas of the project, and this deliverable constitutes a snapshot of the results achieved thus far. D4.2 is meant as a living document that will evolve as implementation progresses. As the final prototypes are delivered in month 24 with D4.4, we will wrap this up, and present its consolidated version as part of the documentation for the prototypes.

In this sense, this document will evolve to adapt to the changes and needs that will be uncovered as further cooperation and integration takes place.

In this version, the deliverable has introduced the guiding principles in the design of the platform, how it's structured and how it can be used by external applications. Additionally, the document reflects the unanimous agreement from all partners regarding the interfaces to be used during the development phase. In order to consolidate the cooperation, we have also shown multiple sequence diagrams that illustrate how the different components integrate with each other to provide the platform functionality.

Finally, Section 8 puts it all together by introducing the use of the Platform components to support the migration of the applications and components under development in WP5.

This document and its future evolutions will be used by WP1 both as feedback to the architectural work and as baseline check for requirements. Additionally, WP6 will further benefit from the presentation of components to plan and perform their testing activities.

Finally, from Work Package 4, we would like to acknowledge the effort and dedication poured into this deliverable by all the Open Partners, which now puts the project well on the road towards a fully functional, integrated Open Platform for Migratory applications.

# 10. BIBLIOGRAPHY

1. **Open partners.** *D1.2: Initial OPEN Service Platform architectural framework.* 2008.

2. **Open partners.** *D3.1: Detailed Network Architecture.* 2009.

3. **Magnet Beyond Consortium.** My personal Adaptive Global NET (MAGNET). [Online] 2008. [Cited: June 15, 2009.] http://magnet.aau.dk/.

4. **R. Olsen, M. Martin.** CMF Tutorial. *CMF Tutorial.* March 2009.

5. **Open partners.** *D4.1: Solutions for application logic reconfiguration.* 2009.

6. **Open partners.**. *D5.1: Initial application requirements and design.* 2008.

# A. APPENDIX: OPEN CLIENT INTERFACE

| void::setRunningStatus (String componentID, String runningStatus) | |
|---|---|
| **String** componentID | A **String** identifying the component intended as target for setting the running status. |
| **String** runningStatus | A **String** object containing the target running status (one of: Running, Paused, Terminated, Busy) |
| Returns: | **void** |
| Set the running status of the specified component.<br>Execution: **asynchronously**<br>Callback at the Open Server Orchestrator Interface: **runningStatusChanged()** | |

| void::reconfigureApplication (String applicationID, Configuration configuration) | |
|---|---|
| **String** applicationID | A **String** object containing the id of the application to be reconfigured |
| **Configuration** configuration | A **Configuration** object describing  the configurations of the given application |
| *Returns:* | **void** |
| Reconfigures an application after some changes in the number and in the components configuration has changed.<br>Execution: **asynchronously**<br>This method can be called both on the source device after the migration of some components has started and on the target device once the migration has finished. | |

| void::migrationTriggerAccepted(String sourceDeviceID, Boolean accepted) | |
|---|---|
| sourceDeviceID | The ID of the device that requested the mgiration |
| accepted | A boolean indicating if the trigger and subsequent migration was successful |
| *Returns:* | Void |
| This method is the callback from OS to OC of the triggerMigration() method. The accepted parameter indicates if the migration was triggered and executed successfully. | |

| boolean::reconfigureNetwork(String deviceID, NetworkConfiguration networkConfiguration) | |
|---|---|
| deviceID | The identifier of the device where network reconfiguration is enforced |
| networkConfiguration | The configuration to be enforced |
| *Returns:* | A boolean indicating if the reconfiguration was successful |
| The function changes the configuration of the underlying network connection of a device as indicated by the networkConfiguration object. This object is registered with the registerNetworkConfiguration() method upon device registration, containing relevant network parameters. The method is called asynchronously by the OS on a OC and the status is returned by the networkReconfigured() method. | |

| void::reconfigureComponent(Component component, ConfigurationInstruction configurationInstruction) | |
|---|---|
| component | The component object describing the component. |
| configurationInstruction | This object provides instructions to a component for reconfiguration. |
| *Returns:* | void |
| This method is part of the OC Interface. It is called by the OPEN Server every time reconfiguration of a component is required. This could be the case if context changes or if components migrate. Reconfiguration includes the replacement of a component instance or change of a component's behavior like described in deliverable D4.1 (5). What kind of adaptation has to be performed in a specific situation and how it is realized is described in the configurationInstruction object. | |

| void:: UIRetrieved(String componentID, UI ui) | |
|---|---|
| componentID | The componentID where the UI originated |
| ui | The user interface that has been retrieved |
| *Returns:* | none |
| This method is offered by the OC Orchestrator and it is called by the OS Orchestrator. It represents the callback for the retrieveUI() method. | |

| void:: setAdaptedUI(UI ui, String componentID) | |
|---|---|
| ui | The user interface that has to be uploaded on the target device |
| componentID | The ID of the component which the ui is associated to |
| *Returns:* | none |
| This method is offered by the OC Orchestrator and it is called by the OS Orchestrator. It is aimed at uploading the adapted user interface onto the target device | |

| void::retrieveState (String componentID) | |
|---|---|
| **String** componentID | A **String** identifying the component intended as target for retrieving the state. |
| *Returns:* | **void** |
| Get the state of the specified components.<br>Execution: **asynchronously**<br>Callback at the Open Server Orchestrator Interface: **stateRetrieved()** | |

| void::setState (String componentID, State state) | |
|---|---|
| **String** componentID | A **String** identifying the component intended as target for setting the state. |
| **State** state | A **State** object carrying the target state. |
| *Returns:* | **Void** |
| Set the state of the specified component.<br>Execution: **asynchronously**<br>Callback at the Open Server Orchestrator Interface: **stateSet()** | |

# B. APPENDIX: OPEN SERVER INTERFACE

| String deviceID:: registerDevice(Device device) | |
|---|---|
| device | The device that will be registered in the OPEN platform |
| *Returns:* | an univocal ID associated by the platform to the registered device |
| This method is offered by the OS Orchestrator and it is called by the OC Orchestrator in order to register a device (with its capabilities) in the OPEN platform. This method is synchronous and returns an univocal ID that the OS associates to the device. Some invalid device IDs could be defined in order to be used as error codes in case of failure. | |

| String applicationID::registerApplication(Application application) | |
|---|---|
| application | The application that will be registered in the OPEN platform |
| *Returns:* | an univocal ID associated by the platform to the registered application |
| This method is offered by the OS Orchestrator and it is called by the OC Orchestrator in order to register an application in the OPEN platform. When an application is instantiated, an application object must be created and registered in the OPEN platform.<br>This method is synchronous and returns an univocal ID that the OS associates to the application. Some invalid application IDs could be defined in order to be used as error codes in case of failure. | |

| String componentID::registerComponent(Component component) | |
|---|---|
| application | The application component that will be registered in the OPEN platform |
| *Returns:* | an univocal ID associated by the platform to the registered component |
| This method is offered by the OS Orchestrator and it is called by the OC Orchestrator in order to register an application component in the OPEN platform.<br>This method is synchronous and returns an univocal ID that the OS associates to the component. Every time this method is called, the OS Orchestrator must update the hasComponent attribute of the Application object associated to the registered component.<br>Some invalid component IDs could be defined in order to be used as error codes in case of failure. | |

| Boolean::unregsiterDevice(String deviceID) | |
|---|---|
| deviceID | The device that will be unregistered by the OPEN platform |
| *Returns:* | True if the unregistration is correctly performed. |
| This method is offered by the OS Orchestrator and it is called by the OC Orchestrator in order to unregister a device.<br>All of the applications and their components running on the device will be unregistered.<br>This method is synchronous. | |

| Boolean::unregisterApp(String applicationID) | |
|---|---|
| applicationID | The application that will be unregistered by the OPEN platform |
| *Returns:* | True if the unregistration is correctly performed. |
| This method is offered by the OS Orchestrator and it is called in order to unregister an application.<br>The application and all of their components will be unregistered.<br>This method is synchronous. | |

| Boolean::unregisterComponent (String componentID) | |
|---|---|
| componentID | The application component that will be unregistered by the OPEN platform |
| *Returns:* | True if the unregistration is correctly performed. |
| This method is offered by the OS Orchestrator and it is called in order to unregister an application component.<br>The component and its sub-components will be unregistered. Depending components will be paused.<br>This method is synchronous. | |

| Device[]::getDevicesSupporting(String[] componentID) | |
|---|---|
| componentID | An array of component IDs that must be supported. |
| *Returns:* | An array of devices that support all of the argument components |
| This method is offered by the OS Orchestrator.<br>Get a list of devices registered on this Open Server which can fulfill the requirements of the components in the parameter. The Open Server will go through the requirements in the component objects and verify that the devices can fulfill it.<br>This method is synchronous. | |

| void::runningStatusSet (String componentID, String runningStatus) | |
|---|---|
| **String** componentID | A **String** identifying the component whose running status has changed |
| **String** runningStatus | The **String** object containing the running status of the requested component. |
| *Returns:* | **void** |
| Notifies the changed running status of the specified component.<br>Execution: **asynchronously**<br>Callback for the Open Client Orchestrator Interface Function: **setRunningStatus()** | |

| void::ApplicationReconfigured(Boolean isReconfigured, String applicationID) | |
|---|---|
| **Boolean** isReconfigured | True if the application has been successfully reconfigured |
| **String** | A **String** object containing the id of the application that has been |

| | |
|---|---|
| applicationID | reconfigured |
| *Returns:* | **void** |
| Notifies the reconfiguration of the given application.<br>Execution: **asynchronously**<br>Callback for the Open Client Interface Function: **reconfigureApplication()**<br>This method can be called both on the source device after the migration of some components has started and on the target device once the migration has finished. | |

| boolean::triggerMigration(String targetDeviceID, String[] componentID) | |
|---|---|
| targetDeviceID | Identifier of the device intended as target for the migration |
| componentID[] | Array of componentIDs to be migrated to target device |
| *Returns:* | A boolean indicating if the migration succeeded |
| This functionality is aimed at triggering the migration of a set of components towards a specific target device. | |

| void::networkReconfigured(String deviceID, boolean isReconfigured) | |
|---|---|
| deviceID | The identifier of the device that enforced reconfiguration |
| isReconfigured | A boolean indicating if the reconfiguration was successful |
| *Returns:* | Void |
| The function is a callback from OC to OS to the reconfigureNetwork() method called from OS to OC. The function indicates from which device it was sent and whether the reconfiguration was successful. | |

| boolean::registerNetworkConfiguration(String deviceID, NetworkConfiguration networkConfiguration) | |
|---|---|
| deviceID | The identifier of the device that registered the network configuration |
| networkConfiguration | A NetworkConfiguration object containing parameters describing a network connection on the device |
| *Returns:* | A boolean indicating if the network configuration registration was successful. |
| The function registers a network configuration from a device in the mobility support module. The network configuration contains relevant parameters describing one possible network connection on the device. | |

| void::componentReconfigured(Component component) | |
|---|---|
| component | The component object describing the component. |
| *Returns:* | void |
| Because the method reconfigureComponent(…) is called asynchronously by the OPEN Server, it needs to know, whether the reconfiguration has been performed successfully. This method is called by a component if it has successfully been reconfigured its behavior according to the instructions given during the reconfigureComponent(…) method call. | |

| void:: retrieveUI(String componentID) | |
|---|---|
| componentID | The component whose UI has to be retrieved |
| *Returns:* | none |
| This method is offered by the OS Orchestrator and it is called by the OC Orchestrator in order to request to get the UI for the specified component. | |

| void:: adaptedUISet(Boolean isUpdated, String componentID) | |
|---|---|
| isUpdated | A Boolean value saying whether the user interface has been adapted |
| componentID | The ID of the component which the ui is associated to |
| *Returns:* | none |
| This method is offered by the OS Orchestrator and it is called by the OC Orchestrator. It represents the callback for the setAdaptedUI() method. | |

| void::stateRetrieved (State state) | |
|---|---|
| **State** state | The **State** object of the requested component. |
| *Returns:* | **void** |
| Returns the State object of the component referenced in a preceding retrieveState() call**.**<br>Execution: **asynchronously**<br>Callback for the Open Client Orchestrator Interface Function: **retrieveState()** | |

| void::stateSet(Boolean isSet, String ComponentID) | |
|---|---|
| **Boolean** isSet | Signals whether the state was changed successfully.<br>**True: Success –** State changed; **False:** Error – State not changed |
| **String** componentID | A **String** identifying the component which state has been changed. |
| *Returns:* | **void** |
| Signals whether a state change requested was executed successfully.<br>Execution: **asynchronously**<br>Callback for the Open Client Orchestrator Interface Function: **setState()** | |

# C. APPENDIX: OBJECT TYPE DEFINITIONS

| Device Object | | | |
|---|---|---|---|
| ID | deviceID | | |
| Type | Device | | |
| It contains all the information needed by the Migration Platform about a specific device. | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| hasName | String | (1…1) | Name of the device (in human readable format) |
| isAvailable | Boolean | (1…1) | True if the device is available for migration |
| hasCapability | Capability | (1…*) | The set of the Device capabilities |
| IsOfClass | String | (1…1) | The class of the device (e.g. mobile, desktop, console, set-top-box etc) |

| Application Object | | | |
|---|---|---|---|
| ID | applicationID | | |
| Type | Application | | |
| It contains all the information needed by the Migration Platform about a specific application instance. | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| hasName | String | (1…1) | Name of the application (in human readable format) |
| hasLaunchCommand | String | (1…1) | Script for launching the application on the target device |
| hasComponent | Component | (1…*) | The components forming the application |
| hasConfiguration | Configuration | (1…1) | The application specific relationships among components together with the information needed for creating and initializing components |

| Component Object | | | |
|---|---|---|---|
| ID | componentID | | |
| Type | Component | | |
| It contains all the information needed by the Migration Platform about a specific component instance. | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| hasComponentDescription | ComponentDescription | (1…1) | A description of the component in terms of input, output, interfaces etc. |

| | | | |
|---|---|---|---|
| hasRunningStatus | String | (1...1) | One of: Running, Paused, Terminated, Busy |
| hasMigrationStatus | Boolean | (1...1) | Migrating or not Migrating |
| hasState | State | (1...1) | Describes the state of the component |
| hasComponentRelationship | ComponentRelationship | (0...*) | A set of relationships with other components |
| belongsTo | ApplicationID | (1...1) | The application which owns the component at a given time |
| hasRequirement | Requirement | (0...*) | A set of minimum needed requirements in terms of software, hardware, network and UI. |
| | | | |

| Component Relationship Object | | | |
|---|---|---|---|
| ID | relationshipID | | |
| Type | ComponentRelationship | | |
| It contains the description of constraints and relationships between the owner component and other components. Multiple relationships are supported. | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| hasComponent | ComponentID | (1...*) | The componentIDs with which the owner component has a relationship |
| hasRelation | String | (1...1) | Relation type between components (dependency, input-output etc) |
| | | | |

| Capability Object | | | |
|---|---|---|---|
| ID | capabilityID | | |
| Type | Capability | | |
| Contains the capability of an entity in the specified category | | | |
| | | | |
| Name | Type | Multiplicity | Description |
| hasName | String | (1…1) | Name of the capability |
| hasValue | String | (1…1) | Value of the capability |
| hasUnit | String | (1…1) | Unit of the capability |
| usesRetriever | Retriever | (1…1) | Indicates if the value of the capability is retrieved by a retriever and thus dynamic |
| isOfCategory | String | (1…1) | One of: Software, Hardware, Network, UI |
| | | | |

| Requirement Object | | | |
|---|---|---|---|
| ID | requirementID | | |
| Type | Requirement | | |
| Contains a component's requirement of a certain capability of the specified category | | | |
| | | | |
| Name | Type | Multiplicity | Description |
| hasName | String | (1…1) | Name of required capability |
| hasValue | String | (1…1) | Required value of the capability |
| hasUnit | String | (1…1) | Unit of the required capability |
| usesComparator | String | (1…1) | One of: less-than, less-than-equal, greater-than-to, greater-than-equal-to, equal-to |
| isOfCategory | String | (1…1) | One of: Software, Hardware, Network, UI |
| *Example of capabilities and requirements:* | | | |
| A device can have a numeric keyboard capability with the hasName="NumericKeyboard", hasValue="true", hasUnit="", usesRetriever="" (as this is considered a static capability) and ifOfCategory="Hardware". An application can require this capability by using the equal properties in the Requirement object, and using the usesComparator="equal-to", as this is a non-measureable requirement.<br><br>Examples of requirements specifications and capabilities will be presented in D5.3 (6) and the functions where comparisons are made between requirements and capabilities (mainly trigger management, migration orchestration and ALR) will be detailed in D3.4.<br><br>Example use-cases of capabilities and requirements is also be seen in the prototypes demonstrating integration between applications and platform functions. | | | |

| NetworkConfiguration Object | | | |
|---|---|---|---|
| ID | NetworkConfigurationID | | |
| Type | NetworkConfiguration | | |
| Contains the connection-specific details of a connectivity option from a device (source) to another device (target) | | | |
| | | | |
| Name | Type | Multiplicity | Description |
| endpointAddress | URL | (1…1) | The socket and protocol to connect to in order to use this connection |
| sourceDevice | Device | (1…1) | The source device of the connection |
| targetDevice | Device | (1…1) | The target device of the connection |
| | | | |

| ConfigurationInstruction Object | |
|---|---|
| ID | configurationInstructionID |

| Type | ConfigurationInstruction | | |
|---|---|---|---|
| This object contains instructions relating to specific component describing how to reconfigure. Mainly two kinds of instructions are coded within this object. The first one instructs the component to change its internal behavior, while the second one instructs the component to change its bindings to other components. The instructions will be coded within an XML document. Therefore we enable the reconfiguration of components written in different programming languages and running on different operating systems. | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| instruction | XML document | (1…*) | Contains a reconfiguration instruction. |

| Configuration Object | | | |
|---|---|---|---|
| ID | configurationID | | |
| Type | Configuration | | |
| The configuration object describing the configurations of an application. As applications are built out of interacting components, the Configuration object defines among others which components are part of the application, the dependencies between these components and how they may change during runtime based for example on context information or migration trigger. | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| - | - | - | Application Specific |

| UI Object | | | |
|---|---|---|---|
| ID | UIID | | |
| Type | UI | | |
| This object contains a whole description/specification of a user interface. Such a UI specification can be described at a concrete level (in this case it will be a XML-based file describing e.g. the interactors that compose the user interface in a platform-dependent way, how they are composed each other within it, the connections between different presentations belonging to the same user interface,.. ), or at an implementation level (in this case it will contain a specification of a UI using a particular implementation language, e.g. XHTML, C#, etc). | | | |
| **Attributes** | | | |
| Name | Type | Multiplicity | Description |
| uri | URI | (1…1) | It contains the specification of the user interface, made available through the provision of its Uniform Resource Identifier (URI). |
| abstractionLevel | String | (1…1) | It contains the information about the type of UI abstraction level considered (e.g. concrete or implementation level) |

| State | | | |
|---|---|---|---|
| ID | N/A | | |
| Type | State | | |
| Contains a component's state | | | |
| | | | |
| Name | Type | Multiplicity | Description |
| belongsToComponent | String | (1…1) | Pointer to the component |
| hasState | String | (1…1) | BLOB containing the application dependent state of the component |
| | | | |