OPEN Project

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

Title of Document:	Migration Service Platform Implementation
Editor(s):	Miquel Martin
Affiliation(s):	NEC
Contributor(s):	All Open Partners
Affiliation(s):	All Open Partners
Date of Document:	March 2010
OPEN Document:	D4.4
Distribution:	EU
Keyword List:	Prototypes, Migration, Platform
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)

OPEN Project

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

ABSTRACT

This deliverable presents the Base Implementation of the Open Migration Service Platform, providing descriptions, requirements, interfaces and interactions for the main implemented components. This document builds on and updates D4.2, and is meant both as documentation of the Base Implementation and as guideline for future developers.

OPEN Project

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

TABLE OF CONTENTS

1. INTRODUCTION

Over the course of the project, OPEN has defined the architecture for the Migration Service Platform and developed its first implementation. This deliverable aims not only at documenting such implementation, but also at providing the general guidelines for future developers to create their own versions.

For this purpose, the document covers the main platform components by providing details on each module's interfaces, inner workings and hardware requirements. Furthermore, the explanation is contextualized by explaining the usage that the OPEN integrated prototypes make of these components.

Please note that this is not a self-contained document, but rather an update from D4.2 [1]. In order to fully understand the inner details of the Migration Service Platform, the reader might want to consult the architecture presented on D1.4 [1] and the platform design proposed in D4.2 [1]. Furthermore, for an exemplified explanation of how application migration is handled across multiple application types, D5.3 [5] provides details on the application design, and D5.4 [6] presents the prototypes developed in the project.

The following sections present the base implementation of the main components of the Migration Service Platform.

2. CONTEXT MANAGEMENT FRAMEWORK

2.1. DESCRIPTION

The Context Management Framework (CMF) component has been described in detail in D4.2 [1]. It fulfills a double role as provider of sensor information and as storage hub for platform information.

In its sensor capacity, CMF modules (dubbed Context Agents) are deployed on the platform and client side. By adding *retriever* plug-ins, each agent is able to understand certain hardware sensors or software APIs, and report *context events* both on a pull (query) and push (subscribe/notify) interface.

As storage hub, the CMF deals in *Entities*, which could be used to store platform data, such as the transitory application state, or even the dynamically updated data for future use, such as discovered devices.

2.2. INTERFACES OVERVIEW

Because of its double role, the CMF presents two separate sets of interfaces. The first handles the extraction of information from the CMF (queries and subscriptions) and the second deals with storing and maintaining in the CMF Storage components.

On the extraction side, we highlight the pull and push interfaces:

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.	

From the storage point of view, insert, update and delete are explained here:

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	

2.3. INTERACTIONS

Platform components interact with the CMF using its XML-RPC interface and the methods detailed in the previous section. A detailed description can be found in the examples provided in D4.2 [1] but we'll focus here on the way information is retrieved (be it from sensors or modules like device discovery).

This uses the synchronous Query method, which returns a QueryResult with the selected information. Additionally, a Subscription can be used, which synchronously returns a subscription ID. Upon successful execution, the CMF will notify the client component provided callback interface of new information, until the client unsubscribes using the subscription ID. Figure 1 shoes in detail how this occurs.

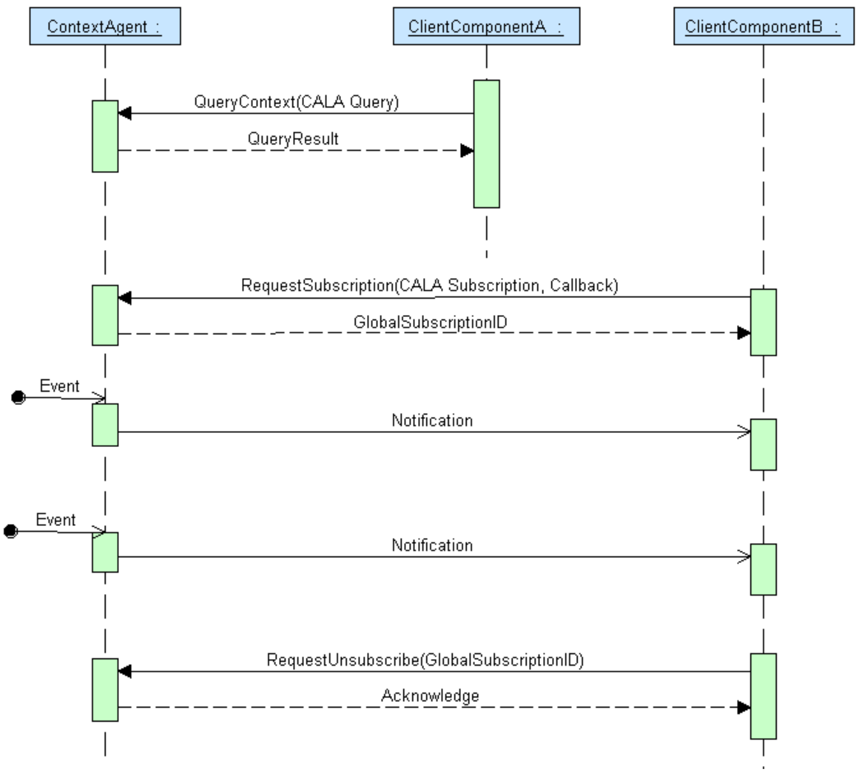

Figure 1 Sequence diagram showing information extraction from the CMF

2.4. HARDWARE AND SOFTWARE REQUIREMENTS

The CMF has been implemented using Java. Furthermore, it has been restricted to the MIDP2/CDC profile, making it suitable to run in very constrained devices, such as Windows Mobile phones. The CMF also runs in Android devices and any hardware above it in terms of computing and memory power. This includes TV Set-top-boxes, Digital Signage modules, tablets and laptop PCs.

Software wise, a Java Virtual Machine is required to run the CMF. Again, constrained devices running CDC JAVA VMs like MySaifu or the Android Dalvik machine, as well as Standard Edition Java and J2EE application servers.

Because of the architecture of the Migration Platform, the CMF does not depend on any other Open component to run, but of course, makes only sense when used by the rest of the components.

2.5. SETUP AND RUN

While we have been referring to the CMF as a component, its deployment actually involves the installation of Context Agents in the different devices.

Installation requires only copying the Context Agent binaries on the hard drive and executing the provided ContextAgent executable. On first run, configuration files will be created in the user's home folder. It will then be necessary to edit the configuration file in order to add or remove sensors as needed.

The CMF is a distributed platform, but is not yet capable of self-organization. For that reason, a Context Agent has to be chosen as a "master" device (referred to as the Context Management Node or CMN), and the rest of the Agents need to be pointed to the URL of the CMN. While any device can be elected as CMN, it is recommended to choose the one with the best CPU performance and bandwidth, given that it will handle a slightly larger message volume, due to synchronization operations between the rest of the Agents.

2.6. USAGE EXAMPLE IN THE PROTOTYPES

The four prototypes described in D5.4 [6] make use of the Context Management Framework. While the complete application usage is detailed in that document, we focus here on the application aspects that rely on the CMF to perform their functions.

TWITTERWALL

The TwitterWall application relies in the CMF both for sensor and device discovery purposes.

A Bluetooth sensor is used to detect the proximity of mobile terminals the public display where the Output component is running. This information is used by the Application Logic Reconfiguration to determine towards what devices the application can be migrated.

An RFID reader is installed under the large public display, and, using the appropriate CMF plug-in, the data is used by the Trigger Management: when the user swipes his RFID enabled mobile terminal on the reader, this is taken as an explicit user interaction to request the migration.

SOCIAL GAME

The Social Game can indirectly use the CMF by requesting devices supporting certain capabilities at the Orchestrator. This then provides a filtered list, of the available devices.

EMERGENCY-PROTOTYPE

The Emergency Scenario also uses the CMF for context sensing purposes. When users approach the shared screen, it is again the swipe of an RFID card that requests the migration of the emergency situation forecast on the display. Note that this could just as well be triggered other context sensors just as easily (e.g. a combination of Bluetooth proximity and a pressure sensor on the floor)

PAC-MAN

The PAC-MAN application uses the CMF for device capability discovery, such as screen resolution. Upon registration, the Open Clients report on their specifications, and these are then used by PAC-MAN to adapt to the migration target.

3. MIGRATION ORCHESTRATION

3.1. DESCRIPTION

The Migration Orchestration module coordinates all of the components required to perform a migration in the OPEN platform. For example, when an application is migrated for a device (the source device) to another device (the target device) the Migration Orchestration module makes sure that the application internal state is correctly transferred from the source device to the target device and that the source version of the application is paused at the beginning of the migration and terminated when the migration is completed. For further details about the role of the Migration Orchestration module in the OPEN Migration Service Platform, please read [D1.4].

The Migration Orchestration module has both a server and a client component. The server side component is available in the OPEN server and it is the main component of the Orchestration. The client component of the Migration Orchestration is an OPEN Adaptor (please read D4.2 for further details about OPEN Adaptors) and there could be several implementations of the module (e.g. a PC version, a mobile phone version, an application-embedded version, etc.). For further details about the Migration Orchestration please read [D3.4].

3.2. INTERFACES OVERVIEW

The methods offered by the Migration Orchestration module can be split in three categories:

- Server Interface. This interface can be used by OPEN applications to take advantage of the features offered by the Migration Orchestration. This interface is offered by the Migration Orchestration server and by the Migration Orchestration client. It is possible to use these methods via XML-RPC.
- Client Interface. This interface is used for the interaction between the Migration Orchestration and OPEN applications. This interface is offered by the Migration Orchestration client and it must be offered by OPEN applications. It is possible to use these methods via XML-RPC.
- Internal Interface. This interface is used to for the interaction between the Migration Orchestration and the other modules of the OPEN Migration Service Platform.

Methods and objects defined in [D4.2] have been updated in order to take into account further implementation details.

SERVER INTERFACE

This interface contains the following method defined in D4.2:

Server Interface	String deviceID:: registerDevice(Device device)
	String applicationID::registerApplication(Application application)
	String componentID::registerComponent(Component component)
	Boolean::unregsiterDevice(String deviceID)

	Boolean::unregisterApp(String applicationID)
	Boolean::unregisterComponent (String componentID)
	Device[]::getDevicesSupporting(String[] componentID)
	void::runningStatusSet (String componentID, String runningStatus)
	boolean::triggerMigration(String targetDeviceID, String[] componentID)
	void::stateRetrieved (State state)
	void::stateSet(Boolean isSet, String ComponentID)

The following methods have been instead added to the interface during the development activity:

Boolean:: abortMigration(String targetDeviceID, Object[] componentID)	
targetDeviceID	The ID associated to the target device of the aborted migration.
componentID	The IDs of the migrated components.
Returns:	TRUE if the migration is correctly aborted.
This method interrupts an ongoing migration and provides the synchronization point to roll it back.	

Application::getApplication(String applicationID)	
applicationID	The ID of the application that will be retrieved.
Returns:	The required application.
This method returns a required application.	

Component[]::getApplicationComponentsOnDevice(String applicationID, String deviceID)	
applicationID	The application ID.
deviceID	The device ID. If deviceId is null or if it is an empty String all of the application components are returned.
Returns:	An array of components.
This method returns the array of components owned by an application and running on the selected device.	

Application[]::getApplicationsOnDevice(String deviceID)	
deviceID	The device ID.
Returns:	An array of applications.
This method returns the array of OPEN applications running on the selected device.	

Component[]::getComponentsOnDevice(String deviceID)	
deviceID	The device ID.

Returns:	An array of components.
This method returns the array of OPEN components running on the selected device.	

Device:: getDevice(String deviceID)	
deviceID	The device ID.
Returns:	A Device object.
This method returns the required Device object	

CLIENT INTERFACE

This interface contains the following methods (please read D4.2 for further details):

Client Interface	void::setRunningStatus (String componentID, String runningStatus)
	void::migrationTriggerAccepted(String sourceDeviceID, Boolean accepted)
	void::retrieveState (String componentID)
	void::setState (String componentID, State state)

INTERNAL INTERFACE

The internal interface offers the following methods defined in D4.2:

Internal Interface	void:: UIRetrieved(String ComponentID, UI ui)
	void:: setAdaptedUI(UI ui, String componentID)
	void:: retrieveUI(String componentID)
	void:: adaptedUISet(Boolean isUpdated, String componentID)

The following methods have been instead added to the interface during the development activity:

void:: triggerMigration(Configuration triggeredConfiguration)	
triggeredConfiguration	A configuration object that represents the new configuration of the application.
Returns:	void
This method is used by the TriggerManagement module to trigger a migration.	

void:: setMode(Component component, String mode)	
component	The component that will be reconfigured.
mode	A String that represents the new mode of the component.
Returns:	void
This method is used by the ALR module to reconfigure an application component.	

3.3. INTERACTIONS

As explained in D3.4, the Migration Orchestration module interacts with the Application Logic Reconfiguration, the Mobility Support and the Trigger Management.

The Application Logic Reconfiguration is used in the following way:

- When an application is registered by the Migration Orchestration, it registers the application and its components in the ALR module.
- The TriggerManagement interacts with the Migration Orchestration (by using the internal interface) to trigger a migration.

In the sequence diagram in Figure 2 it is possible to see the procedure for the registration of an application:

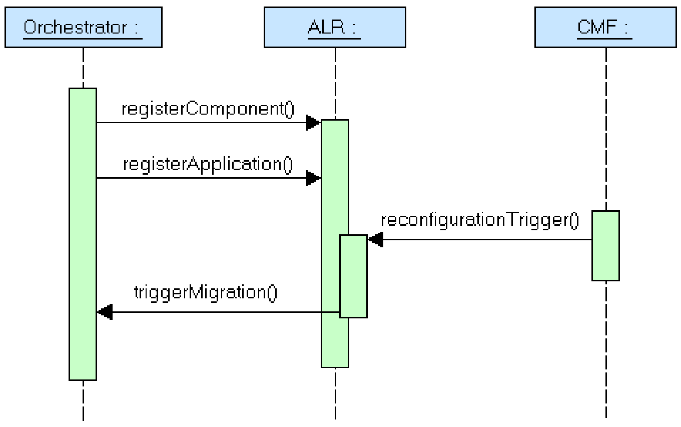

Figure 2: Application Logic Reconfiguration and its interaction with the Migration Orchestration

The Trigger Management uses the Migration Orchestration to trigger the migration of one or more components of an application to a target device.

In the sequence diagram in Figure 3 it is possible to see the usage of the Migration Orchestration during the partial migration of an application with two components (for further details please read D3.4):

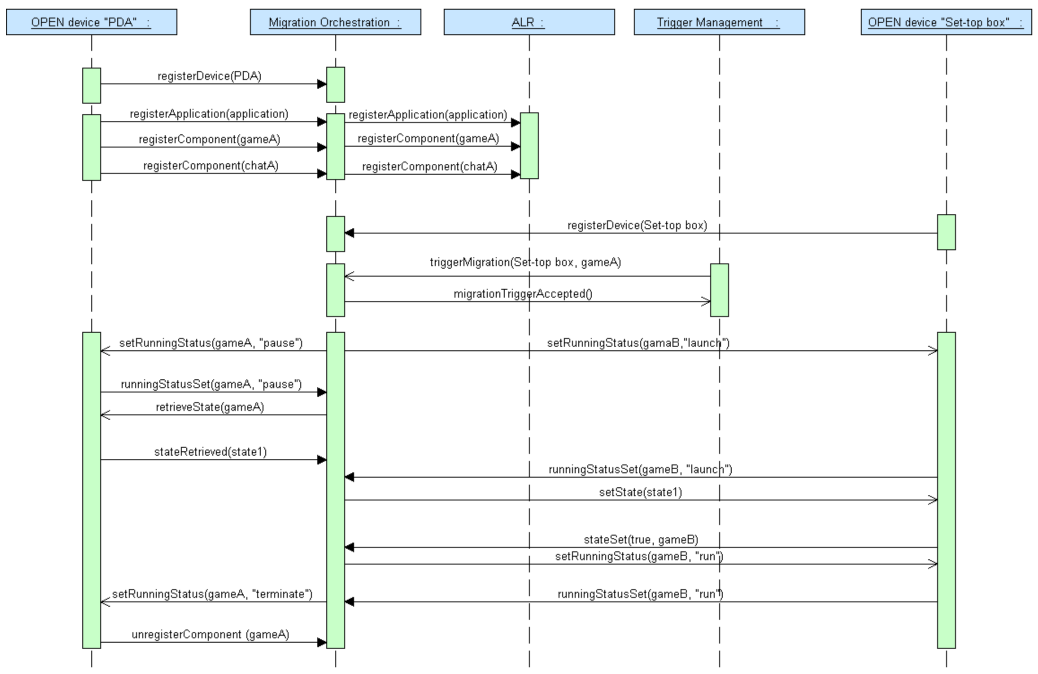
OPEN device "PDA" :
Migration Orchestration :
ALR :
Trigger Management :
OPEN device "Set-top box" :
registerDevice(PDA)
registerApplication(application)
registerApplication(application)
registerComponent(gameA)
registerComponent(gameA)
registerComponent(chatA)
registerComponent(chatA)
registerDevice(Set-top box)
triggerMigration(Set-top box, gameA)
migrationTriggerAccepted()
setRunningStatus(gameA, "pause")
setRunningStatus(gamaB,"launch")
runningStatusSet(gameA, "pause")
retrieveState(gameA)
stateRetrieved(state1)
runningStatusSet(gameB, "launch")
setState(state1)
stateSet(true, gameB)
setRunningStatus(gameB, "run")
setRunningStatus(gameA, "terminate")
runningStatusSet(gameB, "run")
unregisterComponent (gameA)

Figure 3: partial migration of an application from a PDA to a Set Top Box

In Figure 4 it is instead available the sequence diagram of the migration of a generic web application (please read D3.4 for further details):

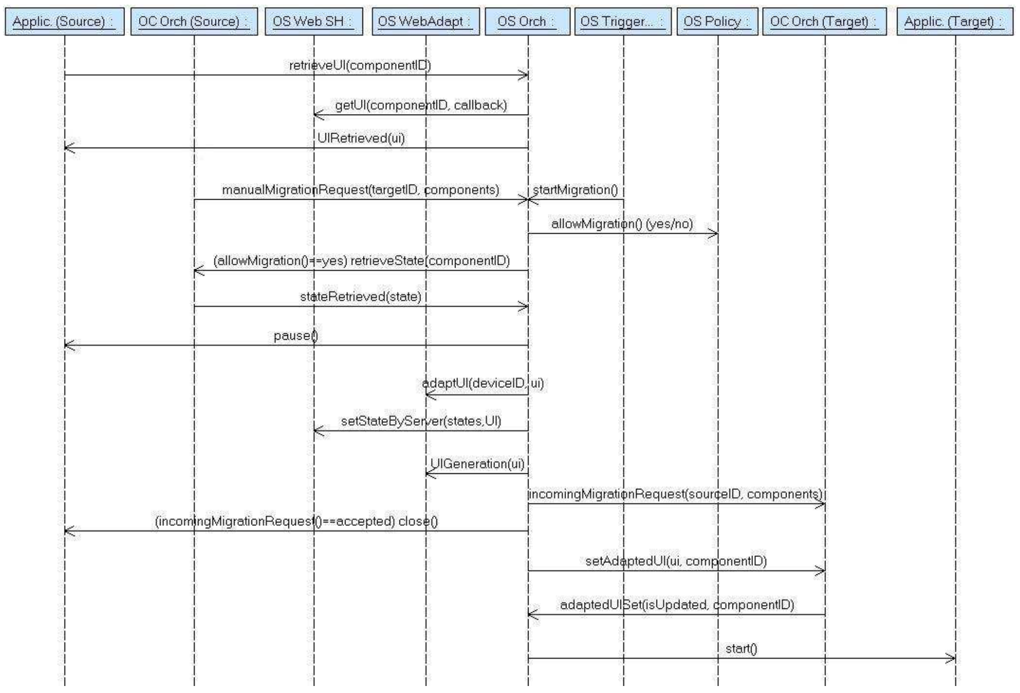
Applic. (Source) :
OC Orch (Source) :
OS Web SH :
OS WebAdapt :
OS Orch :
OS Trigger... :
OS Policy :
OC Orch (Target) :
Applic. (Target) :
retrieveUI(componentID)
getUI(componentID, callback)
UIRetrieved(ui)
manualMigrationRequest(targetID, components)
startMigration()
allowMigration() (yes/no)
(allowMigration()==yes) retrieveState(componentID)
stateRetrieved(state)
pause()
adaptUI(deviceID, ui)
setStateByServer(states,UI)
UIGeneration(ui)
incomingMigrationRequest(sourceID, components)
(incomingMigrationRequest()==accepted) close()
setAdaptedUI(ui, componentID)
adaptedUISet(isUpdated, componentID)
start()

Figure 4: The sequence of communications between the various modules in case of a web migration

3.4. HARDWARE AND SOFTWARE REQUIREMENTS

The Migration Orchestration server has the following requirement:

- A server connected to a network and that uses an IP address accessible by OPEN clients (i.e. a public IP address or an IP address in the same LAN of the OPEN clients).
- Java Virtual Machine (v 1.6). The module has been implemented in java language and it needs a JVM to be run.
- The Application Logic Reconfiguration module. This is needed only if a reconfiguration of the application logic will be performed. If this module is not available, it is not possible to register the application in the ALR and no reconfiguration is performed.

The Migration Orchestration server can have two configurations:

- With an embedded XML-RPC server. In this case, when the module is started, an XML-RPC server is automatically started.
- With Apache Tomcat. It is possible to use the Migration Orchestration as a servlet and to use it with a web container. The version 6.0 of Apache Tomcat has been used.

The implemented PC version of the Migration Orchestration Client has the following requirements:

- A device connected to a network and that uses an IP address accessible by the OPEN server (i.e. a public IP address or an IP address in the same LAN of the OPEN server).
- Java Virtual Machine (1.6). The module has been implemented in java language, and it needs a JVM.

3.5. SETUP AND RUN

If the Migration Orchestration is used with Apache Tomcat, the following section has to be included in the **web.xml** configuration file of Tomcat:

```
<servlet>
  <servlet-name>OpenServlet</servlet-name>
  <servlet-class>open.orchestrator.server.OpenServlet</servlet-class>
  <init-param>
    <param-name>enabledForExtensions</param-name>
    <param-value>true</param-value>
    <description> Sets, whether the servlet supports vendor extensions
for XML-RPC. </description>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>OpenServlet</servlet-name>
  <url-pattern>/xmlrpc</url-pattern>
```

```
</servlet-mapping>
```

The <url-pattern> parameter can be customized according to the URL that will be used to send XML-RPC requests to the Migration Orchestration.

Moreover, the Migration Orchestration must be included in the libraries used by Apache Tomcat.

When the Migration Orchestration is used with its embedded server, it simply needs to be run. XML-RPC requests can be sent to the following URL: http://xxx.xxx.xxx.xxx:8989/xmlrpc, where xxx.xxx.xxx.xxx it the IP address of the server.

The developed Migration Orchestration Client has a java configuration file (config.properties) in the resources folder.

The following parameters have to be set:

- OrchestratorClientPort. The TCP port used by the XML-RPC server embedded in the Orchestration Client.
- deviceName. A name associated to the current OPEN device.
- deviceURL. The URL that can be used by the Orchestration Server to send XML-RPC requests to the Client (usually it is: http://xxx.xxx.xxx.xxx:port/xmlrpc, where xxx.xxx.xxx.xxx it the IP address of the client and port is the value of OrchestratorClientPort).
- OrchestratorServerURL. The URL of the Orchestration server (e.g. http://10.22.72.66:8989/xmlrpc)

Procedure to start the Migration Orchestration:

1. Make sure that the OPEN Server and OPEN devices are connected to the network.
2. Start the Orchestration Server. Use the command “java -jar OrchestratorServer.jar” and wait for the following message: “OPEN Orchestrator Server Ready!”. If the embedded server is not used, it is needed only to start Apache Tomcat.
3. Start the Orchestration Client on every OPEN device. Use the command “java -jar OrchestratorClient.jar” and wait to read the message: “OPEN Orchestrator Client Ready!”
4. Run OPEN applications.

3.6. USAGE EXAMPLE IN THE PROTOTYPES

SOCIAL GAME

The Migration Orchestration has been used for the registration of the devices in the OPEN Migration Service Platform and for the management of a partial migration from a PC to another PC and from a PC to a PDA using the Web UI Adaptation and the Device Selection Map. Both the Migration Orchestration Server and the Migration Orchestration Client have been used. The embedded XML-RPC server has been used.

EMERGENCY MANAGEMENT

The Migration Orchestration has been used for the management of OPEN devices during the migration of flooding and traffic simulations. Moreover, the triggerMigration method has been used to trigger an application migration. In this case only the Migration Orchestration Server has been used (the client-side functionalities were included in the application) and it was used in Tomcat. A pull interface have been developed to communicate with devices that don't implement an XML-RPC server (please read D4.2 for further details about the dispatching of messages in OPEN).

TWITTERWALL

The Migration Orchestration module has been used for the management of automatic migrations. In particular, the output component of a twitter client was migrated. Moreover, the integration with the ALR module has been used for the application reconfiguration.

PACMAN

The Migration Orchestration module has been used for the management of the migration of the Pacman application. A reconfiguration of the application has been performed by using the integration between the Migration Orchestration and the ALR module.

4. APPLICATION LOGIC RECONFIGURATION

4.1. DESCRIPTION

The server side Application logic reconfiguration module (ALR) has been described in detail in D4.2 [1]. It 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.

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).

4.2. INTERFACES OVERVIEW

The ALR computes a list of possible configurations for an application. Therefore, the application has to be registered at the ALR as well as the available application components. The reconfiguration has to be triggered, if the context of the application changes or a new component for an application is available. The following methods are implemented in the ALR module:

registerApplication(Application)	
Application	An application description of the registered application.
Returns:	
Registration of an application at the ALR.	

registerComponent(Component)	
Component	A component description of the registered application component.
Returns:	
Registration of an application component at the ALR.	

unregisterApplication(Application)	
Application	An application description of the registered application.
Returns:	
Deregistration of an application at the ALR.	

unregisterComponent(Component)	
Component	A component description of the registered application component.
Returns:	
Deregistration of an application component at the ALR.	

reconfigurationTrigger()	
Returns:	
Trigger for the reconfiguration. Calculates the result table and notifies the trigger management.	

4.3. INTERACTIONS

As depicted in figure 1 the Orchestrator registers the application and the application components at the application logic reconfiguration (ALR), when they become available.

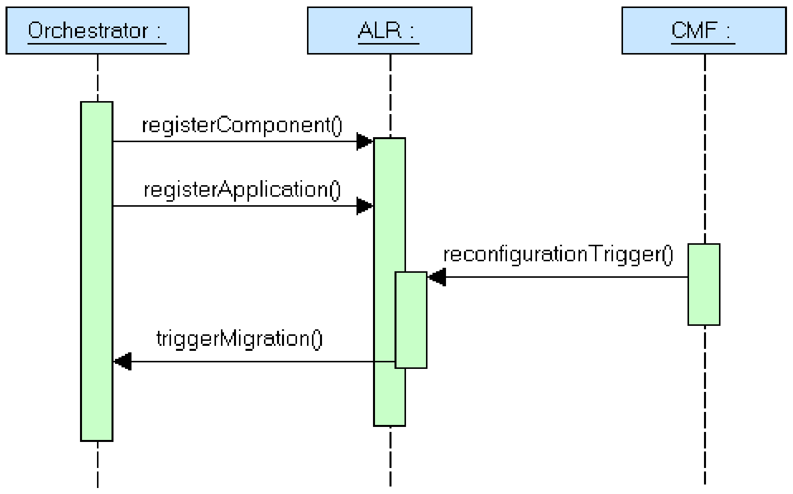

Figure 5: Application logic reconfiguration

The ALR computes a sorted list of configurations based on the registered components and context information. The context information is provided by the context management framework (CMF). The best configuration of the ALR view is the first configuration of the resulting list.

A reconfiguration trigger can be initiated by the CMF caused by an update of a context value. The ALR updates the sorted list of configuration and selects the best configuration out of the ALR view. Afterwards it triggers the migration at the orchestrator.

In cooperation with the trigger management the ALR works as follows. At the start of the open server trigger management (TM) subscribes to scoring functions (e.g. ALR) as depicted in figure 1. Those components are then responsible to notify whenever their output changes. The TM can assume the current output at any time.

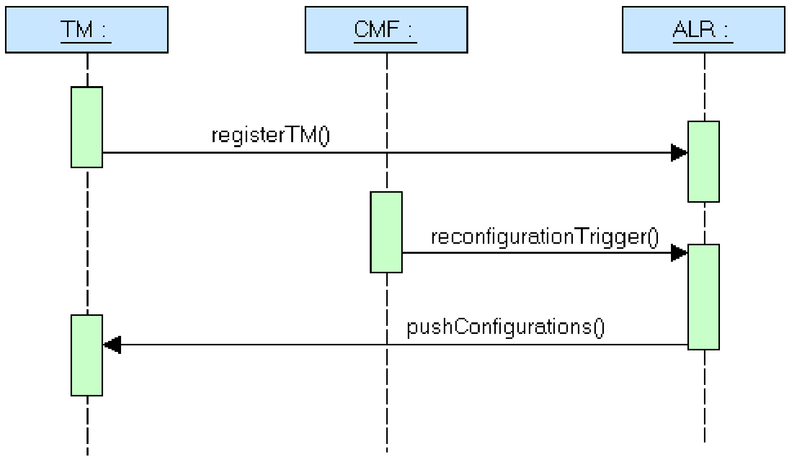

Figure 6: ALR TM interaction

The ALR computes a sorted list of configuration based on the registered components and context information. The context information is provided by the context management framework (CMF). The best configuration out of the ALR point of view is the first configuration of the resulting list. The ALR notifies the trigger management (TM) about changes of the configuration list. The TM accepts the updated list and selects the best configuration based on its scoring functions.

4.4. HARDWARE AND SOFTWARE REQUIREMENTS

The ALR module is part of the Open Server and has the following requirements to the hardware and software:

The device for the application logic / OPEN Client / Open Server has to provide a network card for the web service, the connection from the orchestrator client to the orchestrator server and from the context management framework (CMF) to the clients.

The OPEN platform is implemented in Java. To execute the software at least Java Runtime Environment 1.5 has to be installed.

The ALR works at the operating systems Windows and Linux. Other operating systems are not yet supported.

4.5. SETUP AND RUN

The ALR is included in the OPEN server, which is delivered by an executable java archive called “OpenServer.jar” and a configuration file called “OpenServer.ini”. To setup the OPEN server this files have to be stored in one folder. In the configuration file the following entries have to be updated:

- LocalIp= [the ip of the target platform (server) out of the clients view e.g. 135.34.6.2]

- Port= [the port of the target platform view e.g. 8765]
- OpenServerAddress= [the address of the open server e.g. http://139.0.65.1:8989]
- CmfServerAddress= [the address of the CMF server e.g. http://139.0.65.1:8989]
- WebServiceAddress= [the address of the target platform (server) out of the clients view e.g. http:// 135.34.6.2:8989]

The OPEN Server can be started by the following command:

```
java –jar OpenServer.jar
```

If the operating system supports a direct execute of jar files, the Open server can be started by a double click.

The context management framework (CMF) consists of the file ContextAgent.exe and can be started at the server.

4.6. USAGE EXAMPLE IN THE PROTOTYPES

TWITTERWALL

The TwitterWall application uses the ALR to get the best configuration out of the application view. The ALR considers the distance of the TwitterWall clients to the TwitterWall to compute the order of the configuration table. If a client is near, the TwitterWall is used as output device for this client. The result table with the possible configurations out of the ALR view is forwarded to the trigger management. The trigger management selects the best possible configuration out of the ALR view considering further constraints.

PAC-MAN

The PAC-MAN application uses the ALR to get the best configuration of the application logic. The ALR computes the best fitting ghost logic by considering the current screen size of the PAC-MAN game. If the game is running on a device with big screen size intelligent ghost logic is selected. On a device with small screen size the easy ghost logic is selected.

5. UI ADAPTATION

5.1. DESCRIPTION

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.

5.2. INTERFACES OVERVIEW

The UI Adaptation module has to support the task of retrieving the UI of a certain application component considered, and also to capture the state that has been produced as a result of the interactions that have carried out up to the point when the migration is activated. In addition, it has also to set the captured state of a certain component, and to manage the adaptation of the UI components once a migration has been requested.

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 it has been explained in Deliverable 4.2 [2], the following methods are provided internally by the Web UI Adaptation module.

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)

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::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. Execution: **asynchronously** 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. Execution: **asynchronously** Callback at the Open Server Orchestrator Interface: **stateSet()**	

5.3. INTERACTIONS

Figure 7 shows the migration of a Web Application (see [2]), in which the communications occurring between the various components (also including the Web UI Adaptation module) for supporting such migration are described. In this figure, the application (source device) asks the OS Orchestrator for retrieving the user interface (retrieveUI()) of the specified component(s). Then, the OS Orchestrator asks the OS Web State Handler to get the user interface for the specified components, also specifying an optional callback, which will be only used in case of asynchronous request. Afterwards, there is the callback method UIRetrieved() from the OS Orchestrator to the application (source device). Then, the migration could be triggered either manually from the OC Orchestrator (source device) to the OS Orchestrator (manualMigrationRequest() function), or automatically, coming from the OS Trigger Manager (startMigration() method). After receiving one of such requests, the OS Orchestrator asks the OS Policy for the concerned authorization and, if the OS Policy module permits such migration, then the OS Orchestrator asks the OC Orchestrator (source device) to retrieve the state of the specified component. After the state has been retrieved, the OS Orchestrator pauses the application on the source device and then asks the Adaptation module to adapt the UI. This means to perform the reverse engineering of the retrieved web page, getting a logical UI description of it, and then 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 final UI for the target device. Then, the OS Orchestrator has to asks the OC Orchestrator on the target device the permission to activate a migration on such a device. If the received answer is positive, the application on the source device is closed, and then the adapted UI is uploaded on the target device (setAdaptedUI()). After receiving the callback (adaptedUISet()), the OS Orchestrator can then start the application onto the target device.

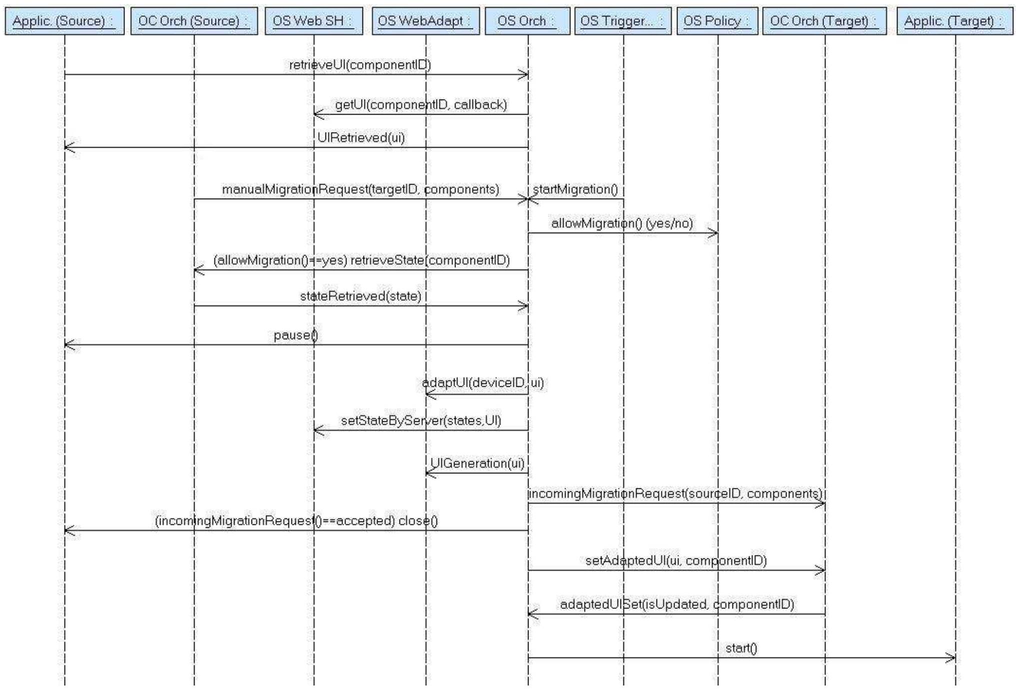
Applic. (Source) :
OC Orch (Source) :
OS Web SH :
OS WebAdapt :
OS Orch :
OS Trigger... :
OS Policy :
OC Orch (Target) :
Applic. (Target) :
retrieveUI(componentID)
getUI(componentID, callback)
UIRetrieved(ui)
manualMigrationRequest(targetID, components)
startMigration()
allowMigration() (yes/no)
(allowMigration()==yes) retrieveState(componentID)
stateRetrieved(state)
pause()
adaptUI(deviceID, ui)
setStateByServer(states,UI)
UIGeneration(ui)
incomingMigrationRequest(sourceID, components)
(incomingMigrationRequest()==accepted) close()
setAdaptedUI(ui, componentID)
adaptedUISet(isUpdated, componentID)
start()

Figure 7: The sequence diagram describing the sequence of activities carried out to migrate a web application

5.4. HARDWARE AND SOFTWARE REQUIREMENTS

Any device running Web browsers can be involved in the user interface migration exploiting this technical solution. There can be some problems with browsers for mobile devices that do not support Ajax scripts. Regarding the migration server, since the UI Adaptation module has been implemented in Java, a Java virtual machine should be available and properly running . Any device satisfying this requirement can act as migration server. However, in order not to add delays connected with the various transformations that are handled by this module, it is recommended to choose a device with an adequate CPU performance. The device has also to run a servlet container (such as Apache Tomcat) in order to expose the embedded proxy server that allows to access and annotate web applications. The migration server automatically enhances the Web application with migratory capabilities (included the partial migration functionality) by validating, annotating and engineering reversing it.

5.5. SETUP AND RUN

The Web UI Adaptation support runs on a server which acts as a servlet container, in our case we used Apache Tomcat. In order to be properly set up, in the “webapps” folder of the root folder of Apache Tomcat, a folder containing the files necessary to support the module has to be included (in our case this folder is called “OpenDemo”). Afterwards, the Apache Tomcat server has to be activated and the page for starting the UI Adaptation server administration has to be opened (within the OpenDemo package this page is a JSP page called “MigrationServer.jsp”). Then, within such a JSP page, the server has to be explicitly started by clicking on the “Start server” button. After having verified that the migration clients on the devices involved in the migration are properly running, in the JSP administration page the information on such devices will appear when such devices have discovered each other through their respective migration clients. From this point it is possible to access a web page and after navigating through it for a while from one device, possibly activate a migration to one of the devices available for migration.

5.6. USAGE EXAMPLE IN THE PROTOTYPES

The Social Game and the Pacman prototypes described in D5.4 [4] make use of the UI Adaptation module. For the detailed application usage you should refer to that document, here we focus on the application-related aspects that rely on the UI Adaptation module support.

The Social Game uses the UI Adaptation module for adapting the UI during a partial migration of the different components of the prototype in a desktop-to-mobile device migration.

Within the Pacman prototype, which has been implemented in a single web page for a desktop platform, the UI Adaptation module has been used for providing an adapted UI of the game, during a desktop to mobile total migration. In particular, the UI Adaptation Module, through a semantic redesign phase is able to calculate the cost associated with the single web page of the game (desktop version) and splitting it taking into account the more limited device capabilities of the mobile device.

6. TRIGGER AND POLICY MANAGEMENT

6.1. DESCRIPTION

The trigger management (TM) prototype is implemented as a periodic loop which evaluates a set application configurations with a set of score functions represented by individual Java methods. Each score method represents either a requirement or a preference which should be taken into account in the decision making e.g. whether the user prefers automatic migration or not. Each method applies a score to each possible application configuration, after which the application configurations with the highest score can be chosen. If the chosen configuration changes a trigger with the new configuration is sent to the orchestrator. If other metrics should be taken into account they would just become another evaluation method.

Three different score levels are evaluated in the prototype.

1. The TwitterWall application defines a blacklist of words which can not be displayed on public displays. This is an application specific requirement. The prototype applies a negative score of 100 if a blacklisted keyword is entered since this has the highest decision priority.

2. The TwitterWall user profiles specify whether they are interested in automatically migrating or not. If a user prefers automatic migration a score of 10 is added to configurations with another device than the current. If the user does not prefer auto-migration a score of 20 is added to configurations with the current device (see below).

3. The application preferences in terms of what configuration is the best seen from the application developer's point of view. In the TwitterWall application output on a large display is considered better then on a small one. The possible configurations are appointed a score from 0 to 9 with the highest score to the best configuration seen from an application point of view. For a more detailed description on how different scoring methods are applied in the concept of TM, refer to D3.4 [3].

The TM prototype also handles a very simple form of policy management. For the TwitterWall application, the TM handles filtering of banned words based on lists set by the user. However, in the prototype, the lists are predefined and cannot be changed dynamically. Whenever the user enters a word this is checked against the list by the TM, and the TM adjusts the migration scores accordingly. One example is if certain words should not appear on displays of public type, the TM would trigger a migration away from such a device, if the application was currently on it and having banned words entered.

6.2. INTERFACES OVERVIEW

The TM prototype only exposes one interface, namely toward the Application Logic Reconfiguration component (ALR). In order to obtain the set of configurations to choose from, a method called "pushConfigurations(Vector<Configuration> validConfigurations)" is exposed which accepts a list/vector of configuration objects. Each configuration is a set of components, networks and devices and which can be interpreted by the ALR as defined in D4.2, appendix C [4]. The configurations have been validated by the ALR so that the TM can choose freely between any of them.

void::pushConfigurations (Vector<Configuration> validConfigurations)	
Vector validConfigurations	A vector of Configuration objects that have been validate as possible to run by the ALR.
Returns:	**void**
Have a set of valid configurations pushed from the ALR to RM. Execution: **asynchronously**	

6.3. INTERACTIONS

Interacting with the TM prototype is simple, as the TM operates almost autonomously.

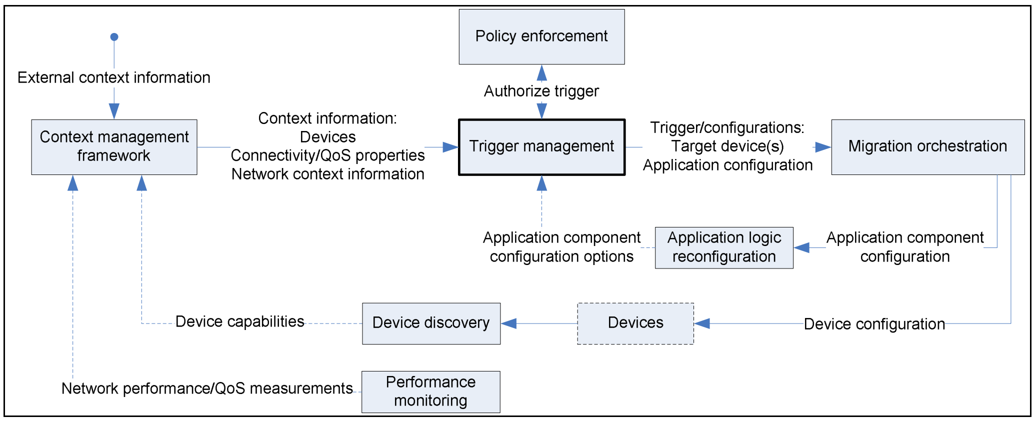
Policy enforcement
Authorize trigger
External context information
Context management framework
Context information: Devices Connectivity/QoS properties Network context information
Trigger management
Trigger/configurations: Target device(s) Application configuration
Migration orchestration
Application component configuration options
Application logic reconfiguration
Application component configuration
Device capabilities
Device discovery
Devices
Device configuration
Network performance/QoS measurements
Performance monitoring

Figure 8: Illustration of component interacting with trigger management

The components interacting with TM are depicted in Figure 8 and the process is described below.

1. Retrieve application requirements (when applications are registered, as a part of the configuration list) from the ALR/Orchestrator
2. Retrieve contextual information from CMF. This can be any relevant piece of environmental information. Information is relevant if it is required by the application.
3. TM makes a decision about which configuration will give the best user experience
4. TM sends the selected configuration to the orchestrator to be effectuated.

6.4. HARDWARE AND SOFTWARE REQUIREMENTS

The TM prototype is fully implemented in Java and can run on any standard Java Virtual Machine. The component is only intended for server-side use, so experiments have not been carried out in order to make it run on resource-constrained devices.

According to the specified interactions, the TM prototype requires the CMF, the ALR and the orchestrator to be installed, and that the offered APIs from the components are known to the TM.

6.5. SETUP AND RUN

The TM is a stand-alone class that is reached through direct method invocation in Java. To start the TM, an object of the class is instantiated. The object then exposes "pushConfigurations()" for receiving configurations.

6.6. USAGE EXAMPLE IN THE PROTOTYPES

The TM prototype was developed for the integrated prototype called "TwitterWall". The functionality reflected in the prototype includes:

- Ability to take application requirements/preferences and user preferences into account.
- A design allowing adding other evaluation parameters without a complete redesign.
- Periodic evaluation of requirements, preferences and changes.
- Trigger use of different application configurations.

The TM prototype revealed different aspects which need to be addressed in a more general TM module for the OPEN platform.

- The score values used in the prototype are not dynamic enough to be used for all purposes e.g. there can not be more than 10 possible configurations to choose from.
- Application configurations are not separated for individual users, meaning both users can "vote"/score for the same configurations. Instead this is solved by using different user scores for automatic migration and non-automatic migration.
- Manual migration triggers are not supported, meaning the user can not force a migration. Applications that require manual triggering can interface directly with the orchestrator.

7. MOBILITY SUPPORT

7.1. DESCRIPTION

The task of the Mobility Support Module is to make migrations of network connections happen transparently to network entities outside of the OPEN network. This is achieved by placing a Mobility Anchor Point (MAP) in the network path between the OPEN client devices and the network entities outside the OPEN network e.g. a streaming server on the Internet. The MAP interacts with the Mobility Support Module on the OPEN server in order to decide how to handle client network connections. The network entities involved in a network migration are shown in Figure 9.

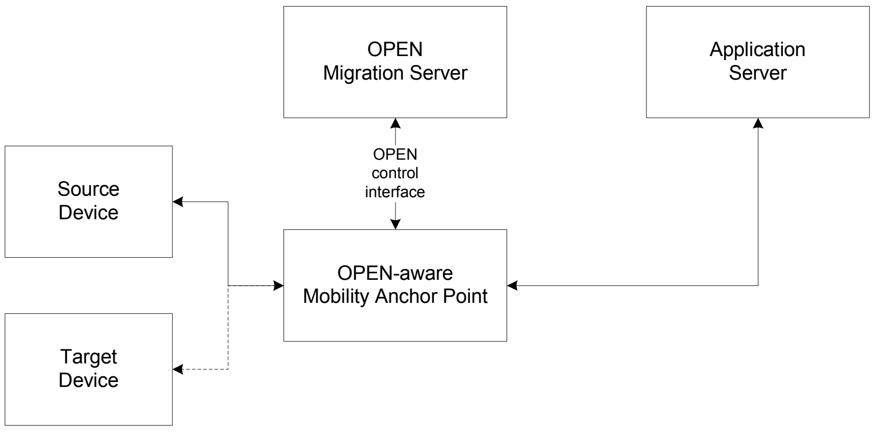

Figure 9 The network entities involved in network migration.

Without the introduction of the MAP the network entities outside the OPEN network would be communicating directly with the OPEN client devices meaning that when a migration occurs e.g. a streaming server should change the network path from the source device to the target device. This is not always possible and instead it has been chosen to let the communication go through the MAP. Doing this the change in network path can happen behind the MAP and e.g. a streaming server will not notice the change. A migration of network connections is achieved by having the target device connect by SOCKS to the MAP and then switch the ongoing network connection from the source device to the target device connection. The mobility support module is discussed in detail in D3.4 [3] and a prototype demonstrating the functionality of the mobility support is described in D3.3 [2].

A migration with mobility support is shown in Figure 10.

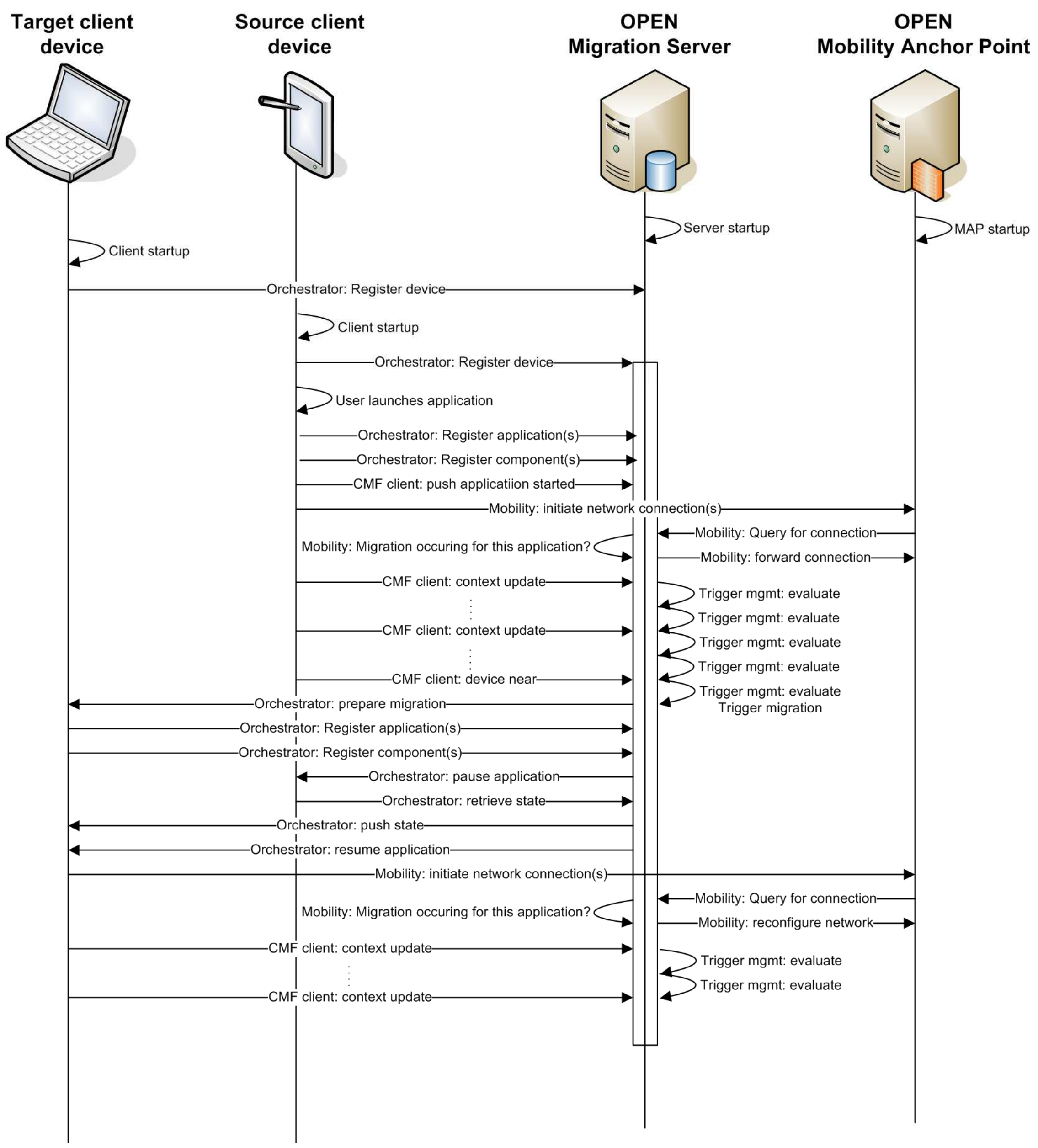
Target client device
Source client device
OPEN Migration Server
OPEN Mobility Anchor Point
Server startup
MAP startup
Client startup
Orchestrator: Register device
Client startup
Orchestrator: Register device
User launches application
Orchestrator: Register application(s)
Orchestrator: Register component(s)
CMF client: push applicatiion started
Mobility: initiate network connection(s)
Mobility: Query for connection
Mobility: Migration occuring for this application?
Mobility: forward connection
CMF client: context update
Trigger mgmt: evaluate
CMF client: context update
Trigger mgmt: evaluate
Trigger mgmt: evaluate
Trigger mgmt: evaluate
CMF client: device near
Trigger mgmt: evaluate
Trigger migration
Orchestrator: prepare migration
Orchestrator: Register application(s)
Orchestrator: Register component(s)
Orchestrator: pause application
Orchestrator: retrieve state
Orchestrator: push state
Orchestrator: resume application
Mobility: initiate network connection(s)
Mobility: Query for connection
Mobility: Migration occuring for this application?
Mobility: reconfigure network
CMF client: context update
Trigger mgmt: evaluate
Trigger mgmt: evaluate
CMF client: context update

Figure 10 Migration with mobility support

The OPEN migration server and MAP have to be started first after which clients can start to use the OPEN migration services. When the OPEN client has registered the device with the migration server, the client application can be started and registered. The component(s) of the application also have to be registered. If the application needs to make a network connection, this will happen though the MAP. The MAP will then query the mobility module on the migration server for what to do with this connection. Since no migration is occurring for this application, the MAP server is told to forward this connection to the application server.

When the application is running, Trigger Management will start evaluating the context information and other factors deciding whether a migration should occur. At some point another better device might become available (it has already registered with the migration server and is in range of the user) and Trigger Management decides to trigger a migration. The Orchestrator will make sure the application is initialized at the target device which makes it register the application and the components. The orchestrator then pauses the application at the source device, extracts the application state and inserts it into the application at the target device. When the application is resumed it will try to use the network connection. This will create a new connection to the MAP and the MAP queries for what to do with the new connection. This time a migration has occurred for the application and the MAP switches the connection from the source device to the target device. The application is now completely migrated an can continue on the target device.

7.2. INTERFACES OVERVIEW

The interface to the mobility support module from the application is defined by the SOCKSv5 protocol since this is the interface to the MAP. It is flexible in the sense that it can either be fulfilled by OPEN middleware on the OPEN client effectively replacing the normal network library used by the application, where the application interface does not change, or the SOCKSv5 protocol can be implemented by the application itself e.g. in Java using the SOCKS library http://jsocks.sourceforge.net/SOCKSLib.html. The prototype of Mobility Support is implemented in Java but because of the SOCKSv5 interface, any language can be used.

Using the first method, the only change in the application, if implemented in Java, will be changing the “import” line to import the OPEN middleware library instead of the usual networking library.

Using the second method the application itself implements the SOCKSv5 interface. Many applications like browsers already implement this. In Java it can be implemented using the SOCKS library. After importing the SOCKS library, information about the SOCKS Proxy (MAP) needs to be inserted into the SOCKS library.

For mobility this is the IP address and port of the MAP server:

Static Void::Socks5Proxy.setDefaultProxy(Hostname, Port)	
Hostname	A String specifying the hostname of the SOCKS proxy
Port	An integer specifying the port number to connect to the proxy
Returns:	Void
Set a default SOCKS proxy	

The MAP needs to pass information about the application, along with the connection information, to the migration server when doing the migration query (See Figure 11). To get this information to the MAP the normal SOCKSv5 authentication is extended. The client should use the following class:

OPENClientAuthentication::OPENClientAuthentication(Username, Password, DeviceID, ApplicationID, ComponentID)	
Username	A String specifying the username used to authenticate to the MAP
Password	A String specifying the password used to authenticate to the MAP
DeviceID	A String specifying the unique device ID for the client device
ApplicationID	A String specifying the unique application ID for the client application
ComponentID	A String specifying the unique component ID for the component
Returns:	OPENClientAuthenticator object
Create an authentication object to use with the MAP	

The authentication is then inserted into the SOCKS library using the following static method:

Static Void::Socks5Proxy.setAuthenticationMethod(MethodID, Method)	
MethodID	An integer specifying authentication method ID
Method	An Authentication object which is the authentication implementation
Returns:	Void
Set the authentication mechanism for the default proxy	

For the OPENClientAuthentication the MethodID should be 10 and the Method should be the OPENClientAuthentication object.

After setting the SOCKS proxy a SOCKS socket can be created if the communication is TCP based:

Socket::SocksSocket(Hostname, Port)	
Hostname	A String specifying the host (application server) to connect the socket to
Port	An integer specifying the port to connect the socket to
Returns:	Socket object
Create a SOCKS socket object	

After the SOCKS socket is created it can be used for network communication like a normal TCP socket.

If the communication is UDP based the following method should be used instead:

Socks5DatagramSocket::Socks5DatagramSocket(Port, IP)	
Port	An integer specifying the port to connect the socket to
IP	A java.net.InetAddress specifying the IP of the host to send data to
Returns:	Socks5DatagramSocket object
Create a UDP SOCKS5 socket object	

7.3. INTERACTIONS

The relationship between the entities involved in mobility is shown in Figure 11.

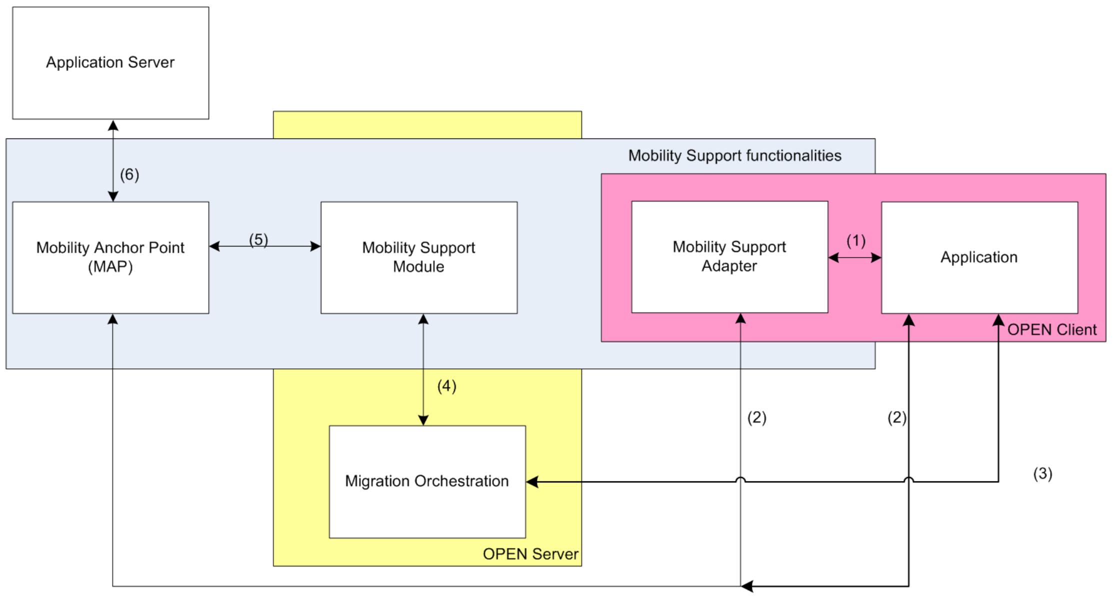

Figure 11 Overview of mobility interfaces. (1 (optional)) The application interface to the OPEN middleware. (2) A SOCKS interface from the OPEN client to the MAP. (3) The application should provide information about the network configuration to the migration orchestration. (4) The mobility support module receives information about migrations involving network changes from the orchestrator. (5) The MAP queries the mobility module about what to do with an incoming SOCKS connection. (6) The interface between the Application Server and the MAP is a normal network interface.

As can be seen in Figure 11, the only other OPEN module which interacts with the mobility support module is the orchestrator. The orchestrator needs to inform the mobility support module that a migration is taking place such that it can answer the connection query from the MAP when the target device makes a network connection.

The interaction between the Orchestrator and the Mobility Support module is detailed in Figure 12. When a migration is triggered the Orchestrator informs the Mobility Support module of the new ApplicationConfiguration. The mobility support module saves the old configuration until a migration query is made from the MAP. The migration query contains information to indentify a connection, including the unique application ID which can be used to look up whether there was a previous configuration for this application. If there was a previous configuration, the connection is from an ongoing migration and the MAP is informed that it should switch from the connection already established.

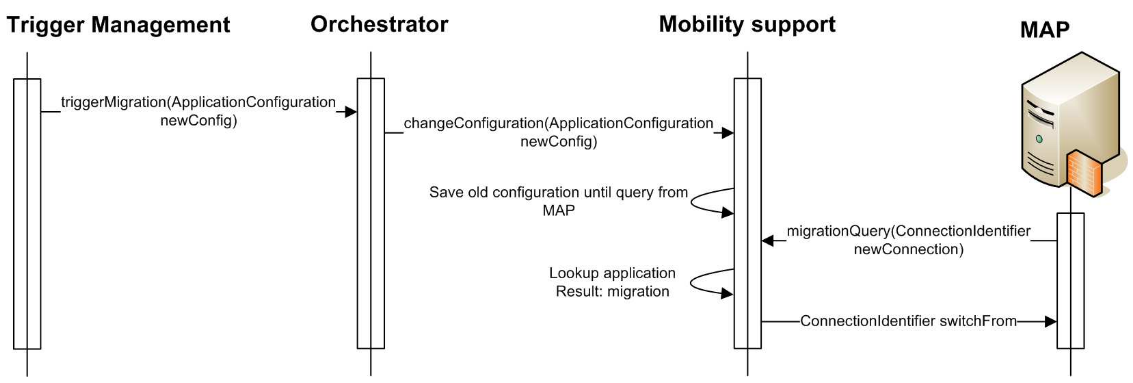
Trigger Management
Orchestrator
Mobility support
MAP
triggerMigration(ApplicationConfiguration newConfig)
changeConfiguration(ApplicationConfiguration newConfig)
Save old configuration until query from MAP
migrationQuery(ConnectionIdentifier newConnection)
Lookup application Result: migration
ConnectionIdentifier switchFrom

Figure 12 Orchestrator interaction with Mobility Support

The interaction between mobility support, with the mobility support adapter installed at the client, and an application is detailed in Figure 13. When an application creates a socket, the mobility support adapter will create a SOCKS connection to the MAP. The MAP will query the MS about the connection, and since it is a new connection it should just be forwarded. When the socket is returned to the application it can start to send and receive data like for a normal socket.

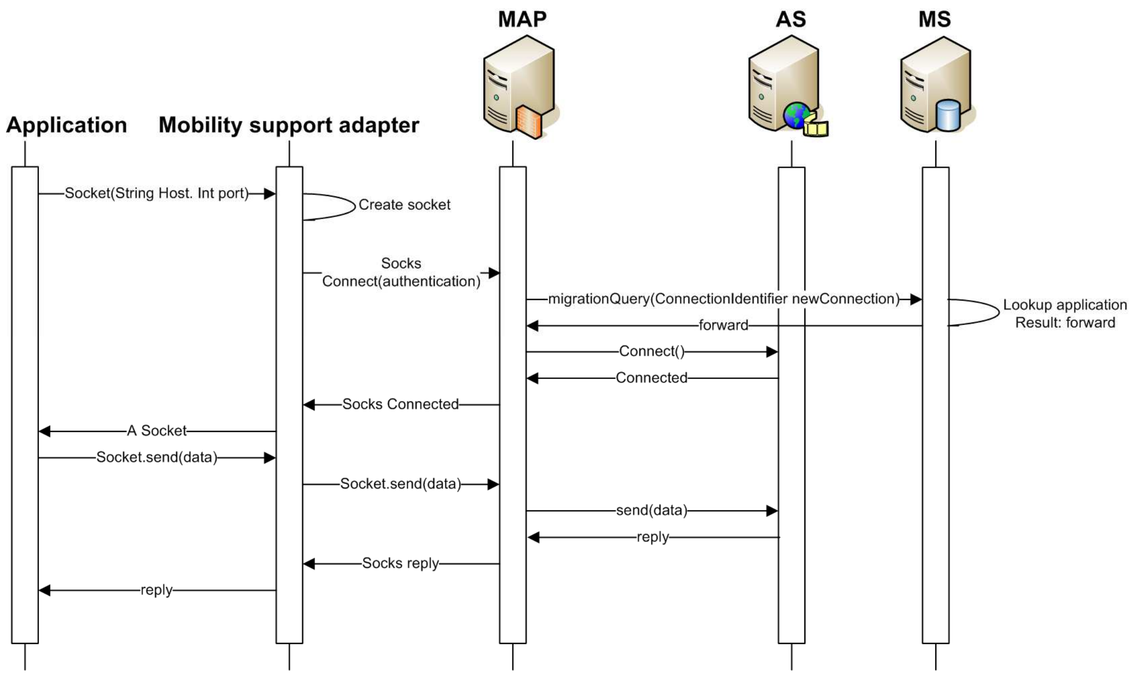

Figure 13 - Application interaction with Mobility Support

7.4. HARDWARE AND SOFTWARE REQUIREMENTS

The prototype of the mobility support functionality has been implemented using Java but could be implemented in any language.

One server is needed for both the OPEN migration server and the MAP, unless scalability requires them to be run on two separate servers. There a no special requirements to the hardware.

Software wise, a Java Virtual Machine on the MAP and the OPEN migration server is required to run the mobility support functionality. Since the interface to the MAP is standardized in the SOCKSv5 protocol the client applications and middleware can be implemented in any language.

The mobility support functionality needs to interact with the orchestrator module in the OPEN migration server.

7.5. SETUP AND RUN

The OPEN migration server and the MAP are normal Java applications that can be started using a Java JRE. The MAP server needs the IP address of the OPEN migration server which can be configured in a properties file. The OPEN clients need the IP address and port of the MAP in order to make the SOCKS connections.

7.6. USAGE EXAMPLE IN THE PROTOTYPES

The only prototype using the mobility support functionality is the Mobility Prototype described in D3.3 [2]. In the Mobility Prototype a streaming application called JOrbisPlayer[1] is the OPEN client application which needs mobility support in order to have a successful migration.

The JOrbisPlayer can play an Ogg Vorbis stream from a http server. If a migration was performed without mobility support, the stream would have to be restarted from the beginning since even with the state transferred it is not possible to start receiving the stream from the server from the point it was left before the migration. Using mobility support the stream can continue on the target device.

[1] Available at http://www.jcraft.com/jorbis/

8. DEVICE DISCOVERY

8.1. DESCRIPTION

The Device Discovery capability is a distributed protocol belonging to the Open Client Daemon, which is a tiny application that runs in background on every Open enabled client. The aim of the Open Client Daemon is to support the user in selecting the target device for migration and in choosing which component(s) migrate. The Client Daemon is available both for desktop PC and for mobile device. The mobile version has an additional feature, the Device Selection Map, that provides the user with a graphical situation of the currently visited environment according, eventually, to user direction and location (see [4]).

The Device Discovery protocol, implemented in C# and executed by any Client Daemon, is able to locally compile and keep a list of currently visible devices (i.e.: potential migration targets) that are shown on the Client Daemon. This is done by multicasting “hello” messages to the peers and by collecting the incoming messages from the peers. The “hello” message contains the identification and a brief description of the sender.

The list of visible devices is kept updated on every Client Daemon by adding newly detected devices and deleting those devices that explicitly leave (that multicast a “bye” message). The local instance of the protocol also assigns a time-to-live to any detected device: every time a “hello” is received, the related time counter is reset. Devices that have not been detected for a long time are removed from the visibility list. This is needed in order to ignore the devices that might have implicitly left (e.g., those that have been abruptly switched off or that are out of connection).

8.2. INTERFACES OVERVIEW

The Device Discovery is not interfaced to any other module of the Open platform.

8.3. INTERACTIONS

Interactions between the Device Discovery and other modules occur only indirectly: information gathered by the Device Discovery is forwarded to the UI Adaptation support, such as the target IP address of the target device during a web migration.

8.4. HARDWARE AND SOFTWARE REQUIREMENTS

The Device Discovery is executed within the Open Client environment. Different versions of the Open Client have been implemented, both for Desktop PCs (Windows XP, Vista, …) and for PDAs (Windows Mobile). The main software need is one of the latest versions of the .NET Framework (such as the 3.5).

Since the Device Discovery protocol is based on message exchange among devices, network connectivity (wireless or wired) is needed.

The devices involved must lie within the same subnet in order to allow multicasting.

8.5. SETUP AND RUN

The Device Discovery setup is contextual to the installation of the Client Daemon: a single executable (i.e.: desktopClient.exe or mobileClient.exe) must be deployed on the device. A XML configuration file must also be deployed. The XML file represents the description that is forwarded to the peers, and thus must be properly compiled with the specifications of the device (name, type, main capabilities).

8.6. USAGE EXAMPLE IN THE PROTOTYPES

Device Discovery is exploited in any prototype that includes the Open Client Daemon, such as the migratory Pacman. In this case, the Device Discovery is run by the Client Daemon and provides the list of target devices for migration.

9. MIGRATION EXAMPLES

This section features sequence diagrams that explain how the migration platform can be used by applications. To achieve this, we show how the OPEN prototypes, which will be demonstrated in the Y2 review, interact at each step.

Figure 14 shows the migration of a component using the OPEN Platform. This migration, for instance, occurs in the Social Game, and shows the message exchanges in full detail. In particular, components register to the platform and, upon triggering a migration from the application, the migration sequence occurs:

1. Pause involved components in the source devices
2. Retrieve the application state
3. Launch the binaries which represent the components in the target devices, and register them
4. Restore the state onto the target devices
5. Terminate the components on the source devices, and run the ones in the target devices.

The case of the Emergency Scenario uses the exact same procedure but also employs the Context Management Framework to generate manual triggers. Using this, both the audio channel and the emergency information are migrated to the big display, as detailed in D5.4 [6]. We do not show a detailed diagram for this, for it would be the same as Figure 14. Instead, we present a sequence diagram for the TwitterWall, which not shows usage of the CMF, but also the Application Logic Reconfiguration and automatic Triggers with the trigger Management.

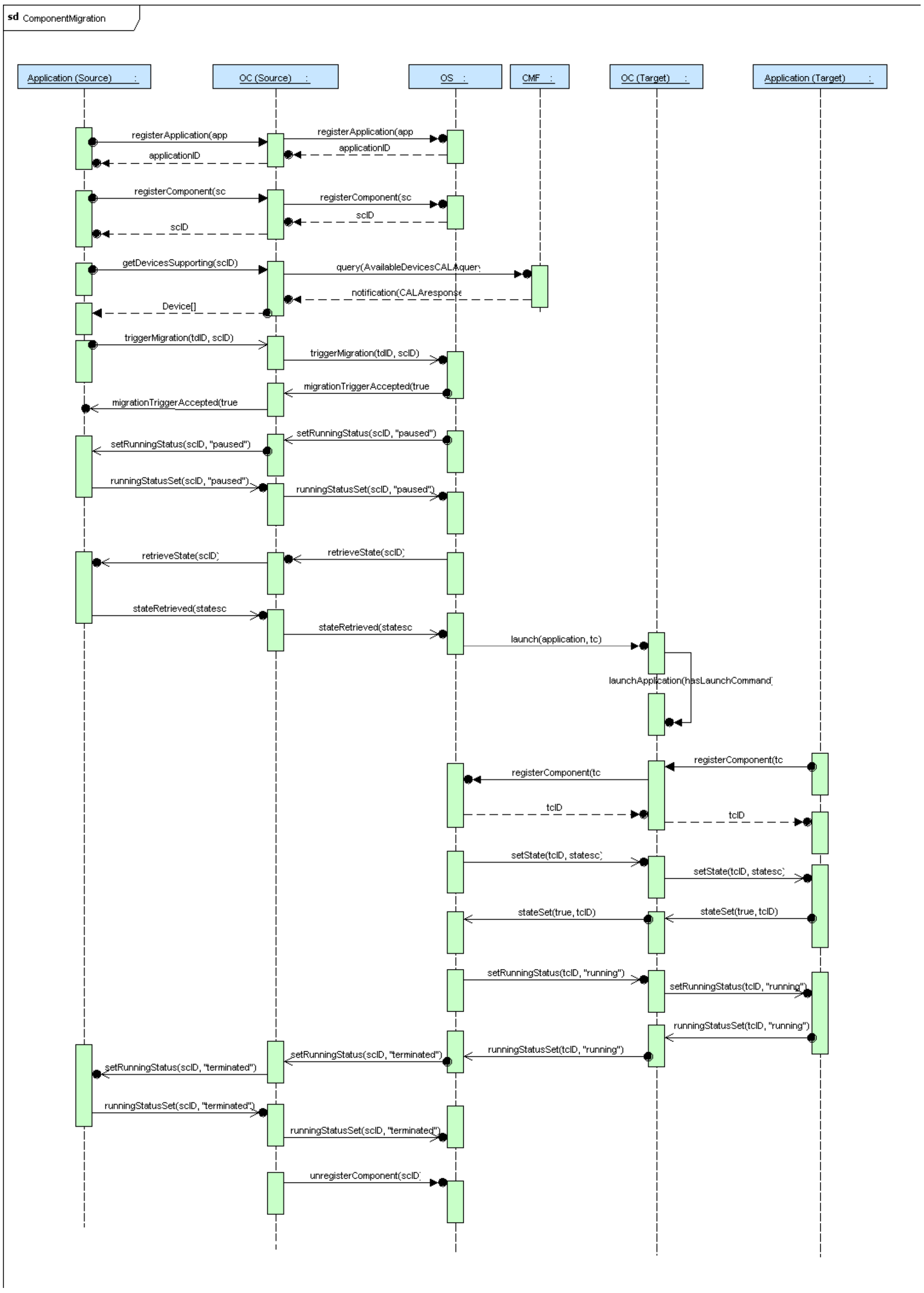
sd ComponentMigration
Application (Source) :
OC (Source) :
OS :
CMF :
OC (Target) :
Application (Target) :
registerApplication(app
applicationID
registerApplication(app
applicationID
registerComponent(sc
scID
registerComponent(sc
scID
getDevicesSupporting(scID)
query(AvailableDevicesCALAquery
notification(CALAresponse
Device[]
triggerMigration(tdID, scID)
triggerMigration(tdID, scID)
migrationTriggerAccepted(true
migrationTriggerAccepted(true
setRunningStatus(scID, "paused")
setRunningStatus(scID, "paused")
runningStatusSet(scID, "paused")
runningStatusSet(scID, "paused")
retrieveState(scID)
retrieveState(scID)
stateRetrieved(statesc
stateRetrieved(statesc
launch(application, tc)
launchApplication(hasLaunchCommand)
registerComponent(tc
registerComponent(tc
tcID
tcID
setState(tcID, statesc)
setState(tcID, statesc)
stateSet(true, tcID)
stateSet(true, tcID)
setRunningStatus(tcID, "running")
setRunningStatus(tcID, "running")
runningStatusSet(tcID, "running")
runningStatusSet(tcID, "running")
setRunningStatus(scID, "terminated")
setRunningStatus(scID, "terminated")
runningStatusSet(scID, "terminated")
runningStatusSet(scID, "terminated")
unregisterComponent(scID)

Figure 14 Migrations in the Social Game

Figure 15 shows the migration as it occurs in the TwitterWall demonstrator. The steps are consistent with those of the Emergency Scenario and the Social Game, so we skip some of the obvious reply messages and all component registrations.

The diagram, however, shows how the TwitterWall Input+Output is terminated, and in its place, two components (one on the source and one on the target device) are started. This is therefore a partial migration, where only the output functionality is migrated away from the source device and into a big display.

Additionally, the CMF is used to detect Bluetooth devices, which are considered as new devices, and therefore result in the Application Logic Reconfiguration preparing a new configuration proposal to the Trigger Management. Note how the first proposal is rejected (on the grounds of privacy), while the RFID swipe convinces the Trigger Management to trigger the migration (i.e. the user input scoring function tips the scales and the new ALR proposed configuration is executed)

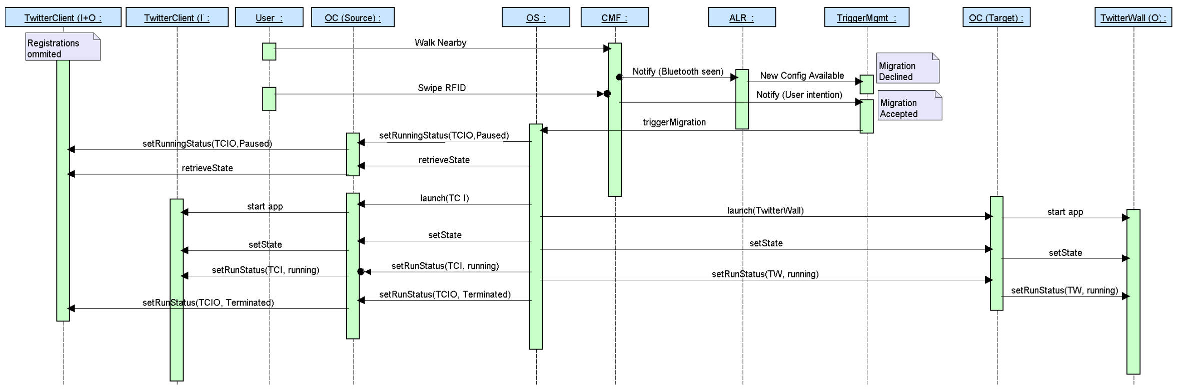

Figure 15 Partial migration in the TwitterWall

While Figure 14 showed the basic migration sequence for non-web applications, Figure 16 shows the case for web based ones (excluding Rich Internet Applications, like the Emergency Scenario). In this approach, the browser obtains a copy of the web application that has been adapted by the Web Migration module. Through the injection of JavaScript scripts, the OPEN Server can retrieve the state of the Web page, and migrate (including re-adaptation) to another browser in a different device.

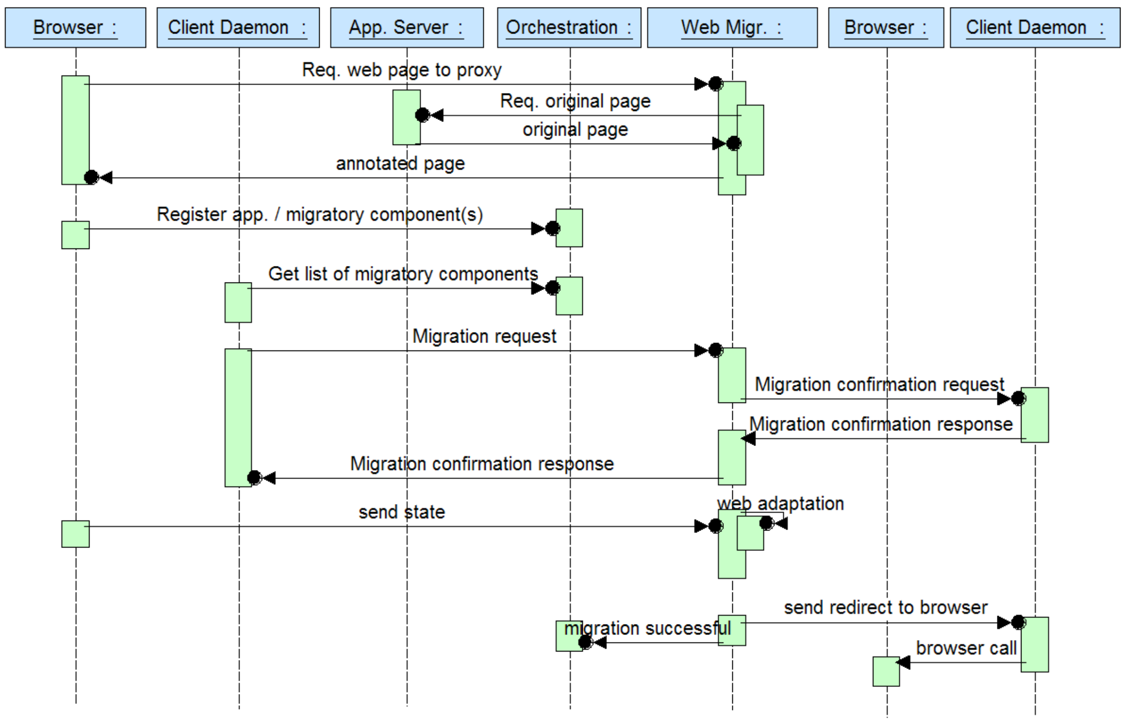

Figure 16 Migration of Web based applications

While Figure 16 shows the standard migration steps for Web applications, Figure 17 expands the explanation by covering a migration that relies on Application Logic Reconfiguration and the Context Management Framework. In this case, the CMF is used to retrieve device information (e.g. screen size) and the ALR is used to adjust the game parameters.

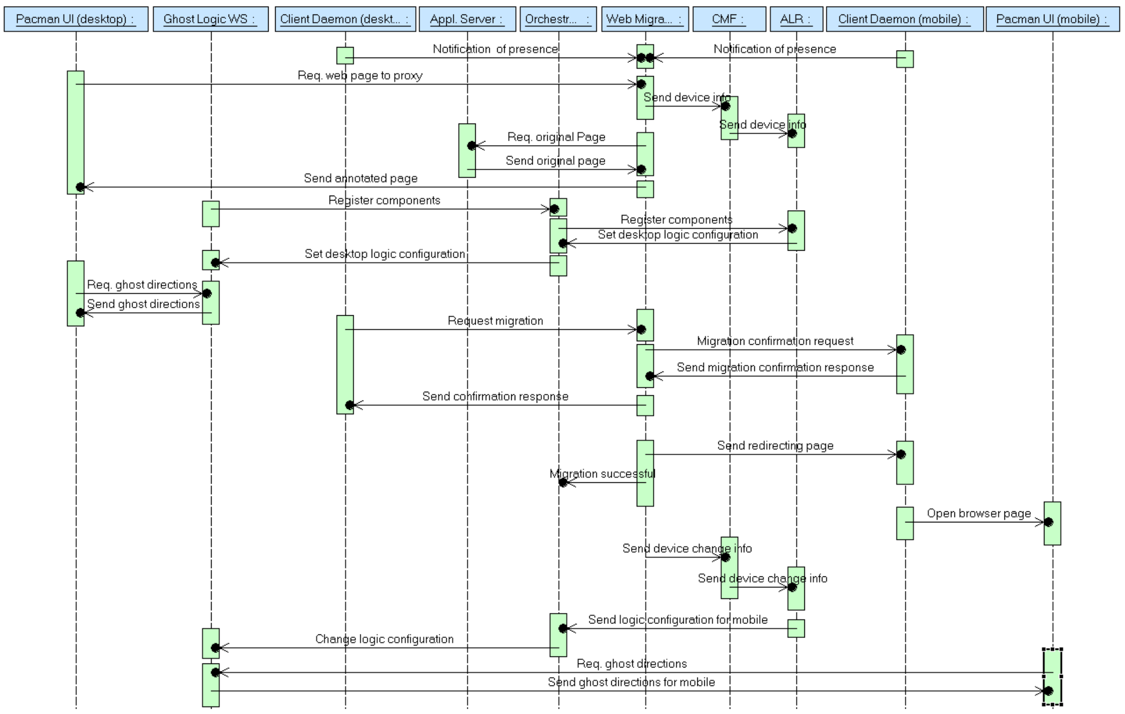

Figure 17 Migration for Web Applications: the Pacman application

10. CONCLUSIONS

This deliverable has detailed the OPEN Base Implementation of the Migration Service Platform. The contents here presented aim to improve the understanding of the development done throughout the project.

More importantly, however, the overview provided here, combined with the details in D4.2, will remain as a guideline for future developers working on their own versions of the Migration Service Platform. Together with deliverable D5.4, future implementers will have both the examples and the platform they rely on, thus helping to further the use of the work developed in the OPEN Project.

11. REFERENCES

[1] " Final OPEN Service Platform architectural framework”, EU FP7 ICT project OPEN, Deliverable D1.4

[2] “System Support for application migration (revised)”, EU FP7 ICT project OPEN, Deliverable D3.3

[3] “Final communication and context management solution for migratory services”, EU FP7 ICT project OPEN, Deliverable D3.4

[4] “Migration Service Platform design”, EU FP7 ICT project OPEN, Deliverable D4.2

[5] “Final application requirements and design”, EU FP7 ICT project OPEN, Deliverable D5.3

[6] “Final prototype applications”, EU FP7 ICT project Open, Deliverable D5.4

A. APPENDIX: OBJECT TYPE DEFINITIONS

The following objects have been defined or changed during the development of the Migration Platform, after the definitions in D4.2 [4] have been frozen

ApplicationConfiguration			
ID	applicationConfigurationID		
Type	ApplicationConfiguration		
The object contains information about devices and components forming one possible application configuration. An application configuration is a realization of an application.			
Attributes			
Name	Type	Multi plicity	Description
configurationNum	Integer	(1…1)	Used for TriggerManagement ordering
belongsToApplication	String	(1…1)	ApplicationID of the application this configuration is for.
neededComponents	Vector<String>	(1…1)	List of ComponentIDs needed to run this application
deviceComponentMap	Map<String,String>	(1…1)	What component(s) is/are on what device(s)
tmScore	Integer	(1…1)	TriggerManagement assigned score
Connection	ConnectionIdentifier	(1…1)	A network connection

ConnectionIdentifier			
ID	N/A		
Type	ConnectionIdentifier		
It contains information about one unique connection made to the Mobility Anchor Point			
Attributes			
Name	Type	Multiplicity	Description
client_addr	InetAddress	(1…1)	IP address of the client
server_addr	InetAddress	(1…1)	IP address of the application server
client_port	Integer	(1…1)	Port number of client
server_port	Integer	(1…1)	Port number of application server
deviceID	String	(1…1)	ID of the device this connection belongs to
applicationID	String	(1…1)	ID of the application this connection belongs to
componentID	String	(1…1)	ID of the component this connection belongs to