OPEN Project

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

Title of Document:	Middleware for multi-user migratory interfaces
Author(s):	G.Ghiani, F.Paternò, C.Porta, C.Santoro, F.Zaccaro
Affiliation(s):	CNR-ISTI
Date of Document:	September 2010
OPEN Document:	D2.6
Distribution:	EU
Keyword List:	Migration, User Interface, Multi-Device Environments, Adaptation, Continuity, Multi-user Applications
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)

ABSTRACT

This deliverable describes the software architecture of the updated prototype implemented for supporting user interface migration. It is a middleware aiming to support automatically the main functionalities: adaptation and state persistence in Web application, across multiple devices with various interaction resources. Among the features of the new prototype there is the support for multi-user applications, which we describe in the document with an accompanying example.

TABLE OF CONTENTS

1. INTRODUCTION

This deliverable is the updated version of the migration software prototype that was presented in D2.5. This is a prototype deliverable, thus the purpose of this text is just to provide an introduction to the demo that will be shown at the final review meeting. This new version of the prototype extends the previous architecture in two main directions: possibility to support partial migration through which the user can directly select the parts of interests, and support for migration in multi user applications. We still consider Web applications without posing any particular requirement on how they should have been implemented.

Indeed, our approach aims to provide a general solution for Web applications implemented using (X)HTML, CSS, and Javascripts. It can also support applications based on languages such as JSP, PHP, ASP because it considers one page at a time on the client side. Thus, it adapts only what is actually accessed by the user. Another advantage of the solution proposed is that it makes Web applications migratory regardless of the authoring environments used by developers. Thus, without requiring the use of any specific tool in the development phase, it enables the applications to migrate, even if developers never considered migration. This is obtained through the use of reverse engineering techniques that create on the fly the logical descriptions of the Web pages accessed, which are then adapted for the target device. After this step, an implementation of the migrated page with the updated state of the page accessed in the source device is dynamically generated on the target platform.

2. WEB MIGRATION IMPROVEMENTS

2.1. GENERAL IMPROVEMENTS

In the following sections we describe the updates that have been done on the Web migration support, which has been improved in several aspects.

2.1.1. WEB-BASED MIGRATION CLIENT

The previous version of Migration Client was developed in .NET C# and could run only in Windows or Windows Mobile –based devices. This requirement was considered a limitation. Thus, a new Web Migration Client has been designed and integrated in the Web migration support, enabling the user to access the migration capabilities directly from the browser.

The web-based Migration Client has been developed in HTML + JavaScript (Ajax) and can potentially run on any device that supports such technologies. Then, differently from the previous version, the web-based Migration Client does not require the installation of any additional software: the only requirement for the user is to have (or to create on the fly) an account on the migration platform and to specify the features of the devices that s/he wants to involve in the migration. In order to show that the web migration can now involve also non-Windows devices, the migration support has been tested also with devices such as iPhone and iPad.

Besides the aforementioned compatibility aspects, executing the Migration Client on a browser also brings performance benefits. Indeed, an important advantage of having *within the same browser instance* both the Migration Client and the page the user is currently accessing, is the possibility of exploiting the inter-window communication between the associated windows. Such mechanism, provided by multi-window/multi-tab browsers, enables a script included in a window to access the content and the functionalities of any child window created from the window that hosts the script. In our case this has been used to enable the window associated with the Migration Client (the 'parent' window) to access and communicate with the ('child') window containing the page that the user is currently navigating (via Proxy). As we will see later on, this access is needed for e.g. enabling the Migration Client window to inform the window containing the concerned web page that a migration has been triggered, as well as to access data and functions defined in the window associated with the web application.

Such inter-window communication is exemplified in the following script:

(from window A)

window_B_reference = window.open(…);

window_B_reference.document.element.attribute = new_value;

…

window_B_reference.function_of_window_B();

This script is supposed to be included and executed within the parent window (window A), and enables to open a new (child) window (window B). The script of *window A* can fully access the document and the functions of *window B*. This includes the possibility of reading/modifying the DOM as well as invoking functions defined in *window B*. The main requirement for the above code to work is that the location of the child window is in the same domain of the parent window. Since the Migration Proxy, through which the various web pages are navigated, is in the same domain as the Web Migration Client, the requirement is fulfilled.

By exploiting this mechanism, the migration platform is now able to overcome the intrinsic lack of communication existing between the previous version of the Migration Client (which was implemented in C# as a standalone application) and the browser displaying the Web application. Indeed, in the migration process, a communication between the Migration Client and the source page is needed for triggering the migration. The migration trigger, generated in the Migration Client, has to "reach" the browser in order to "wake up" the script method that sends the DOM of the source page (together with the current state) to the Migration Server. The latter will elaborate such data and, if needed, adapt it to the target device (it happens e.g. in the case of a desktop-to-mobile migration).

In the previous version of the architecture, the standalone application of the Migration Client and the web page currently accessed in the browser communicated through the Migration Server via Ajax calls and TCP messages through the following (exemplified) steps:

1. The source page, upon loading, called asynchronously a "waiting" servlet (called *Check-Migration Servlet* and deployed on the Migration Server) that responded only when a migration was requested;
2. The Migration Client, upon migration triggering, sent a TCP message to the *Unlock Manager* on the Migration Server specifying the source and the target device;
3. The *Unlock Manager* on the Migration Server "unlocked" the waiting servlet.

The above steps enabled the source web page to detect a migration trigger (activated by the user through the Migration Client). After such a detecting, the source page made an Ajax call to provide the Migration Server with the current DOM. In Figure 1 there is a diagram summarising the communications that took place (in the previous version of the migration support) between the client device and the Migration Server to enable a migration request. In this figure, the (1), (2), (4) and (5) steps are needed by the source page just for detecting an outgoing migration request that has been triggered from the Migration Client (which is running on the same device but in a different context, because it is a separated application). Thus, just for detecting a migration trigger, two inter-device communications took place: an asynchronous Ajax call ((1) / (5)), and a TCP message (3).

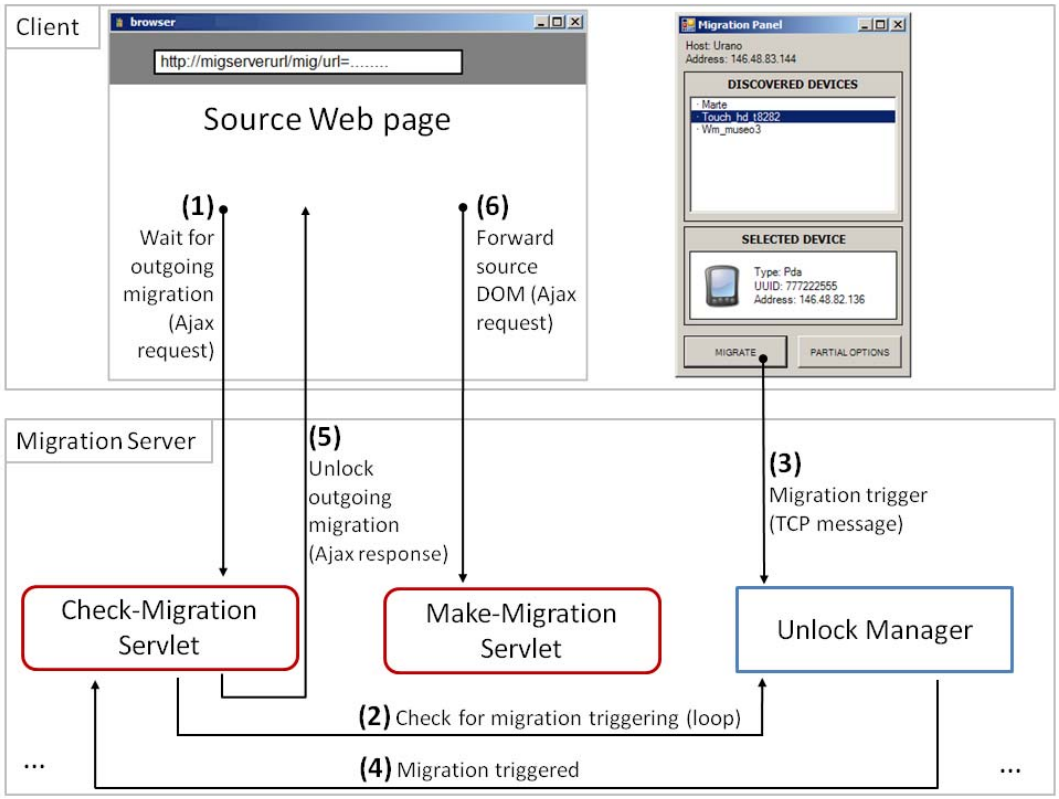

Figure 1: The architecture of the old strategy for detecting outgoing migration requests

Figure 2 shows the new solution for detecting outgoing migration requests. The window from which the user navigates the page (via the migration Proxy Server) is created by the window that contains the Migration Client. The mechanism (described before) based on the unlocking of the waiting servlet is no longer needed and, thus it was removed from the architecture. This reduction of architecture complexity provides the following benefits:

- No need for instantiating a servlet (i.e.: the previously called “Check-Migration Servlet”) for every navigated page, and thus less memory usage at server side;
- Lower network usage;
- Lower latency on migration triggering, since the communication between the Web migration client and the source page is handled within the browser (and not by the Migration Server).

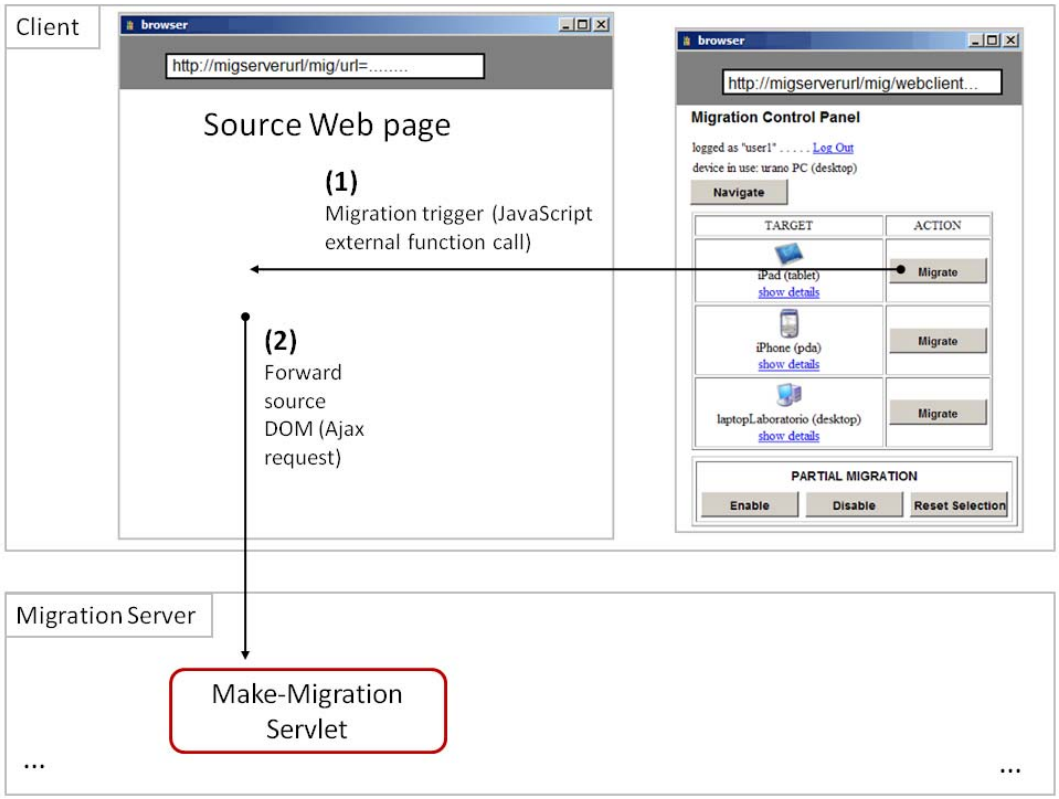

Figure 2: The new solution (simplified architecture) for detecting outgoing migration requests

In general, by reducing the usage of resources (like network and memory) and the latency, the platform performance improves, and this is particularly true when multiple users require access.

2.1.2. DIRECT SELECTION OF COMPONENTS FOR PARTIAL MIGRATION

In the previous version of the migration platform, partial migration was supported through selecting the component(s) to migrate from a tree-view representation of the structure of the page concerned, which was presented within the Migration Client. In this previous version of the support, once the user selected one or more components from the tree-like representation, the Migration Client notified the source page through Ajax (and via the migration server), and the selected components were highlighted. However, this selection modality was affected by two main issues:

1) The difficulty for the user to find, in the list shown in the migration client, the component(s) to select, since they were identified just by names. Then, the reason lied in the meaninglessness of such names, which were automatically assigned;
2) The latency of the client-server communication needed to support the selection-and-highlight mechanism. Indeed, such a communication consisted of a first communication between the Migration Client and the server (to send information about the selected components) and of a second communication between the server and the browser (to highlight the selected component(s) in the source web page);

In order to overcome the mentioned problems, the new support for partial migration was based on the idea of i)providing the user with means for directly selecting the components within the web page (in order to improve the intuitiveness of the approach) and ii) offering a client-based mechanism to provide the user with a quick and intuitive feedback regarding the components currently selected.

In order to do this, the new support for partial migration is based on automatically annotating all the meaningful components of the source page concerned. This means considering all the components that can potentially be subject to migration (e.g. “DIV”, “TABLE”, “FORM”, ..) so leaving out the elements that are basically related to formatting (like
, <p>, etc.). Then, the annotations consist of onclick/onmouseover/onmouseout attributes added to the abovementioned source page elements, so that once such events are detected on the components, the background colour of the component changes. For example, *onmouseover* turns the background to gray colour, *onclick* turns it to green (or to the original one, when the component is deselected).

With this solution, the selection events are entirely handled locally (in the browser). Therefore the set of selected components is highlighted much quicker than with the previous support, since no network communication is used and therefore no network latency occurs. The list of selected components is sent to the Migration Server only when the migration is actually requested by the user.

In addition, from the support of the web Migration Client, it is now possible to enable / disable the partial migration feature, as well as to reset the current set of components selected. The first feature (enable/disable partial migration) was judged useful since the user might want to disable partial migration so as to freely interact with the page (e.g., when filling a form) before migrating, and at the same time avoid a continuous change in the background color of the hovered components. The second feature (reset the current set of selected components) was included in order to enable the user to reset the list of components and start from the beginning in an easy and quick way.

2.1.3. JAVASCRIPT STATE PERSISTENCE

All the JavaScript variables included in the source page should preserve their state in the target page. With regard to the problem of capturing the state represented by JavaScript variables, a key role is played by the Window object, which represents the Web browser window. This object also defines the program namespace, since every JavaScript expression implicitly refers to the current window. Therefore, when JavaScript code contains a definition of a global variable, e.g. “var k=0;”, actually, this is the same as writing “window.k = 0;”. Then, every object/variable defined in the global environment of the JavaScript code is simply a property of the global Window object. However, since we cannot assume to know in advance the names of such user-defined window properties we have to do it programmatically.

To this aim a useful JavaScript construct is the for..in statement, which has the following syntax [6]: for (variable in object) { code to be executed }

where variable is the name of a variable, or an element of an array, or a property of an object.

The *for.. in* statement provides a way to look at the enumerable properties of an object (through the *propertyIsEnumerable()* method that every object provides), and it is exploited within the *saveState()* method used in the migration support to capture the state of the global JavaScript variables. In particular, the statement (i) belongs to the JavaScript code included within the concerned web page by the Proxy Server when the page is downloaded from the application server, and which is executed when a migration is triggered.

```
var jsState = JSStateMigrator.saveState();  (i)
```

After executing the statement (i), the state of the JavaScript global variables is contained in the *jsState* variable, through the *saveState* function, whose key code is described below:

```
saveState(){                                    (ii)
        var globalJSONs = {};
        for (var prop in window) {
            JSONobject = dojox.json.ref.toJson(window[prop], true);
            globalJSONs[prop] = JSONobject;
            …
        }
   return  JSONobject;
}
```

As it can be seen from the (ii) code above, the properties of the *window* object are first listed (with the *for..in* statement) and then saved in a JSON object, which is basically a string. This string is used to update the corresponding element in the global associative array (*globalJSONs*), which at the end will contain all the values of the JavaScript global variables. In order to translate every global variable into a JSON string, a specific library was used (dojox.json.ref, see [3]). The *toJson* function available in such a library creates a JSON serialization of every property of the window object, and therefore it produces an array of couples (<global_variable_name>, <global_variable_content>) for each JavaScript global variable included by the programmer.

Within the script automatically included for supporting migration, after the statement (i) there is an AJAX request directed to the Migration Server. Through such a request, the client passes to a servlet (which is on the Migration Server) both the *DOM* of the page and the *jsState* variable (which contains the current values of the JavaScript global variables). According to such input data, the servlet creates the corresponding page. This means that the servlet identifies the <body> element of the page derived from the received DOM. Then, it appends to the current content (if any) of the *onload* attribute of the <body> element a new JavaScript function called "*restoreState*()". The goal of the latter function is to update the JavaScript global variables

contained in the page, by exploiting the *jsState* parameter (which contains a serialization of the JavaScript global variables). This function is described through the code excerpt below (iii):

```
restoreState {                          (iii)
    // parsing JSON state
    var globalJSONs = dojox.json.ref.fromJson(JSONstate);
    for (name in globalJSONs) {
        try{
            value= dojox.json.ref.fromJson(globalJSONs[name]);
            win[name] = value;
        }
        catch .. {}
}
```

The *restoreState()* function exploits the "*fromJson*" method provided by the dojox library, which de-serializes the state contained in the globalJSONs variable. Such function, when executed, updates the global JavaScript variables that are properties of the *window* object. Then, the servlet i)stores the updated page (containing the up-to-date state) in a specific location of the server and ii)stores in a file the JSON object received from the AJAX script. Depending on the location where such information has been saved, the server is able to build the corresponding URL from which the browser in the target device should load the page with the updated state.

2.2. WEB MIGRATION MULTI-USER SUPPORT

In order to introduce multi-user support we had to improve the session management, which is useful to identify the access from various users. In addition, we had to improve the architecture implementation in order to have distinct management of the files and resources dedicated to each user. In this section we describe how the session management has been sued and then a multi-user scenario that benefits from the support of migration.

2.2.1. SESSION MANAGEMENT

The migration process, at the server layer, creates and manipulates of several resources. Such resources include the temporary files used by the modules of the platform to exchange data (input HTML page, logical interface descriptions, JavaScript state persistence, …) and the generated files (adapted images, target pages and style sheets).

The collection of temporary files begins when the user firstly accesses the Migration Proxy Server. Thus, to enable the multi-user capability, separate repositories are created for each user session.

The Migration Proxy Server exploits the *session-id* of the http request for distinguishing the requests coming from different users and creates temporary session folders for hosting the resources. The *session-id* is a string generated (in an almost random manner) by a Java servlet of the Migration Proxy Server. Such a string (which is composed of 32 characters) is forwarded to the client browser as a value of a cookie named "JSESSIONID". Providing that the browser supports cookies, it will append the session-id cookie to any subsequent request to the Proxy. In this way, the Proxy will be able to recognise the client by checking the value of the session-id cookie contained in the header of the incoming http request. The session-based strategy for client identification works as long as the client browser keeps the session cookie saved within the cache. If the browser cache is cleared and/or the cookies are deleted, the browser will not append any cookie on the http request, and thus the Proxy Server will generate a new session-id and will bind it to the user.

In addition, it is worth pointing out that there is a N:1 relationship between the session-ids and the related user. Indeed, on the one hand a session-id is associated with only one user. On the other hand, a single user can have more than one session active with a server. In our migration support this happens when the same user uses different browser instances at the same time for navigating different pages: in this case the user accesses the Proxy Server through different sessions. The advantage of having a user navigating multiple pages through different instances of browsers is that by using different instances of browsers two or more pages can be migrated from a single source device (e.g., a desktop) to the same target device (e.g., a PDA) or to more target devices (e.g., a PDA and a tablet).

2.2.2. AN EXAMPLE

As an example of multi-user migration we consider two users who are chatting together (using a chat application implemented in HTML, JavaScript, and PHP).to decide which books to buy on the Amazon web site they are visiting meanwhile. In particular, both users (Giuseppe and Carmen) are using the same type of platform (the desktop) as a source device while are using two different browsers (respectively Firefox for Giuseppe and Google Chrome for Carmen) In Figure 3 you can see the chat as it is displayed on the desktop screen of the first user (Carmen), whose name and picture are displayed on the bottom-left part of the chat panel.

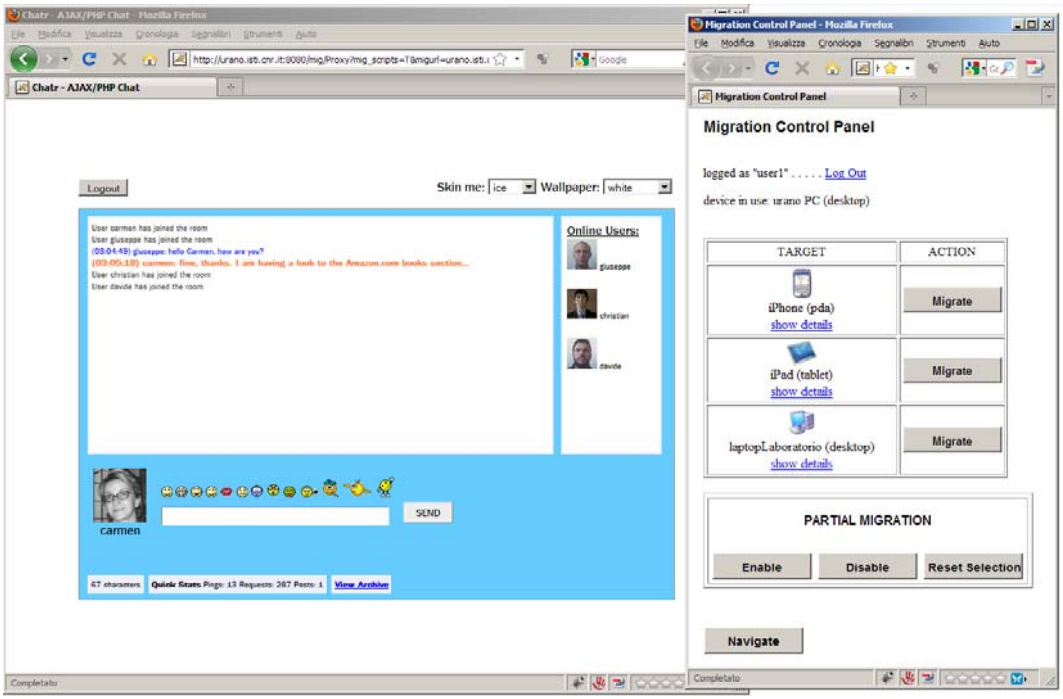

Figure 3: First user (Carmen), desktop device, chat application and Web migration client (Firefox browser)

Likewise, the second user (Giuseppe) is currently talking with the first user through its corresponding chat window in his desktop source device (see Figure 4):

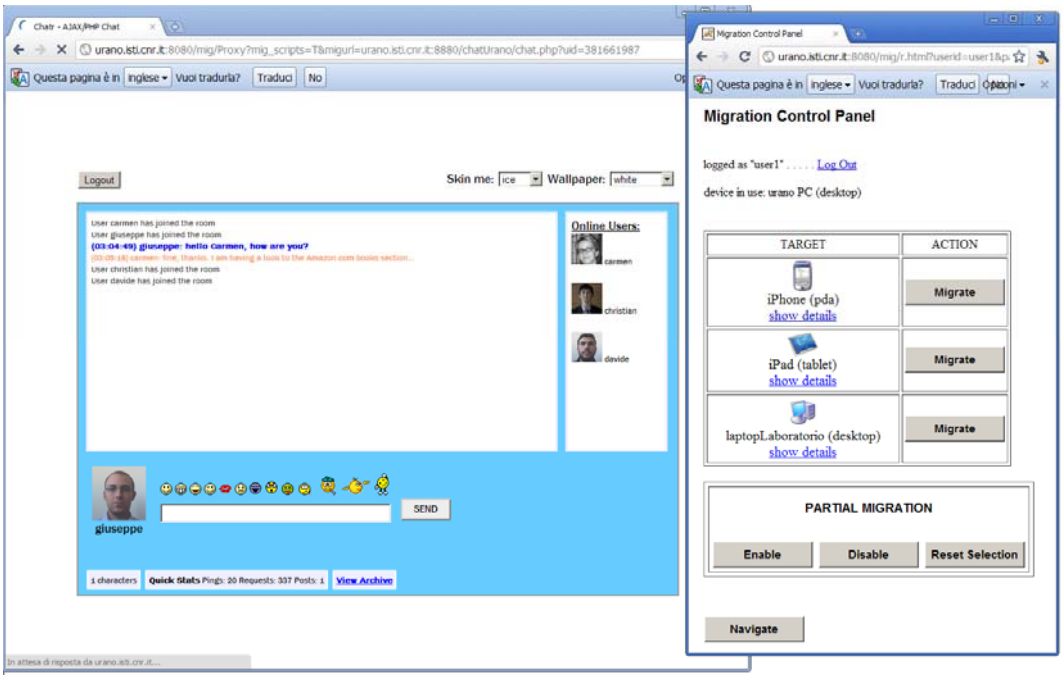

Figure 4: Second user (Giuseppe), desktop device, chat application and Web migration client (Google Chrome browser)

Carmen, in the meanwhile, is looking at the book page of the Amazon web site, shown in Figure 5.

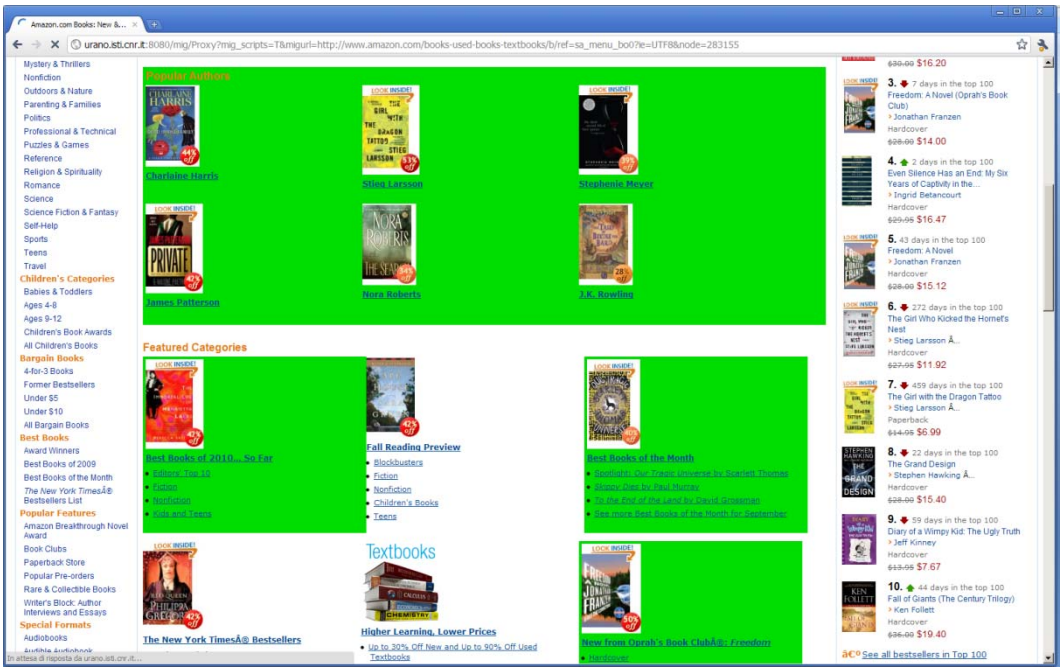

Figure 5: Carmen, desktop device, Amazon application (Google Chrome browser) with components selection for partial migration.

While chatting, the two users decide which book(s) to buy, and then they migrate both their chat desktop application and their Amazon desktop page towards two different mobile devices: an iPhone (Carmen) and an iPad (Giuseppe).

In Figure 6, the window of the Migration Client asking for incoming confirmation for migration can be seen.

Figure 6: Carmen, iPhone device, Web migration client and incoming migration request

After Carmen has accepted the migration, the desktop web page visualizing the chat is migrated towards her target device (iPhone): in Figure 7 you can see the result.

Figure 7: Carmen, iPhone device, chat application migrated

Figure 8 shows the result of the window when Carmen rotates the iPhone device (and then the orientation of the window changes).

Figure 8: Carmen, iPhone platform, chat application migrated (horizontal orientation)

Figure 9 and Figure 10 show the same windows for Giuseppe. In particular, Figure 9 shows the window (visualized on the iPad) asking the user whether he wants to accepts the incoming migration.

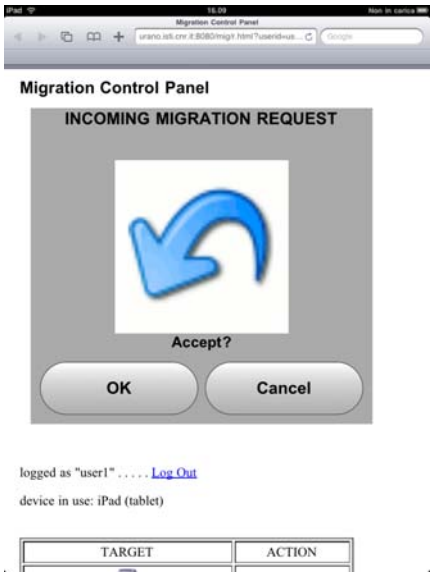

Figure 9: Giuseppe, iPad device, incoming migration request and Web migration client

In Figure 10 you can see the result of migrating the chat on the iPad (Giuseppe).

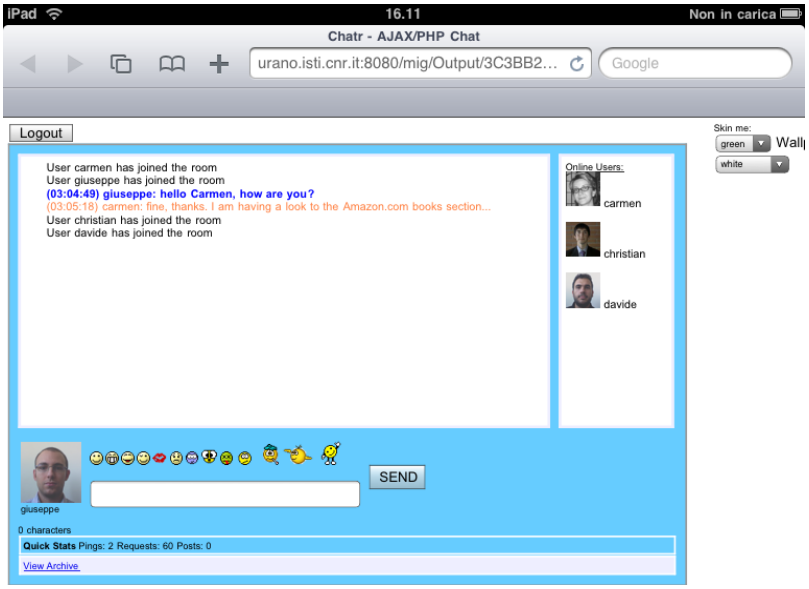

Figure 10: Giuseppe, iPad device, chat application migrated

After migrating the chat towards the iPhone and the iPad, Carmen and Giuseppe also migrate the Amazon page.

In particular, Carmen migrates to her iPhone, which has a relatively small display: then, she prefers to bring to the target device only the components she has previously selected (see Figure 5 for selection details). The adaptation step performed by the Migration support performs a splitting step and delivers two resulting pages visualised in Figure 11 (first page of the splitting) and Figure 12(second page of the splitting).

Figure 11: Carmen, iPhone device, Amazon application partially migrated (page 1)

Figure 12 Carmen, iPhone device, Amazon application partially migrated

Giuseppe, instead, still wants to look at the Amazon book page as a whole. Since he is migrating to the iPad, which has a sufficiently big screen, he triggers a total migration. Thus, the whole Amazon page is adapted by the Migration Server and sent to the iPad: the resulting page is shown inFigure 13.

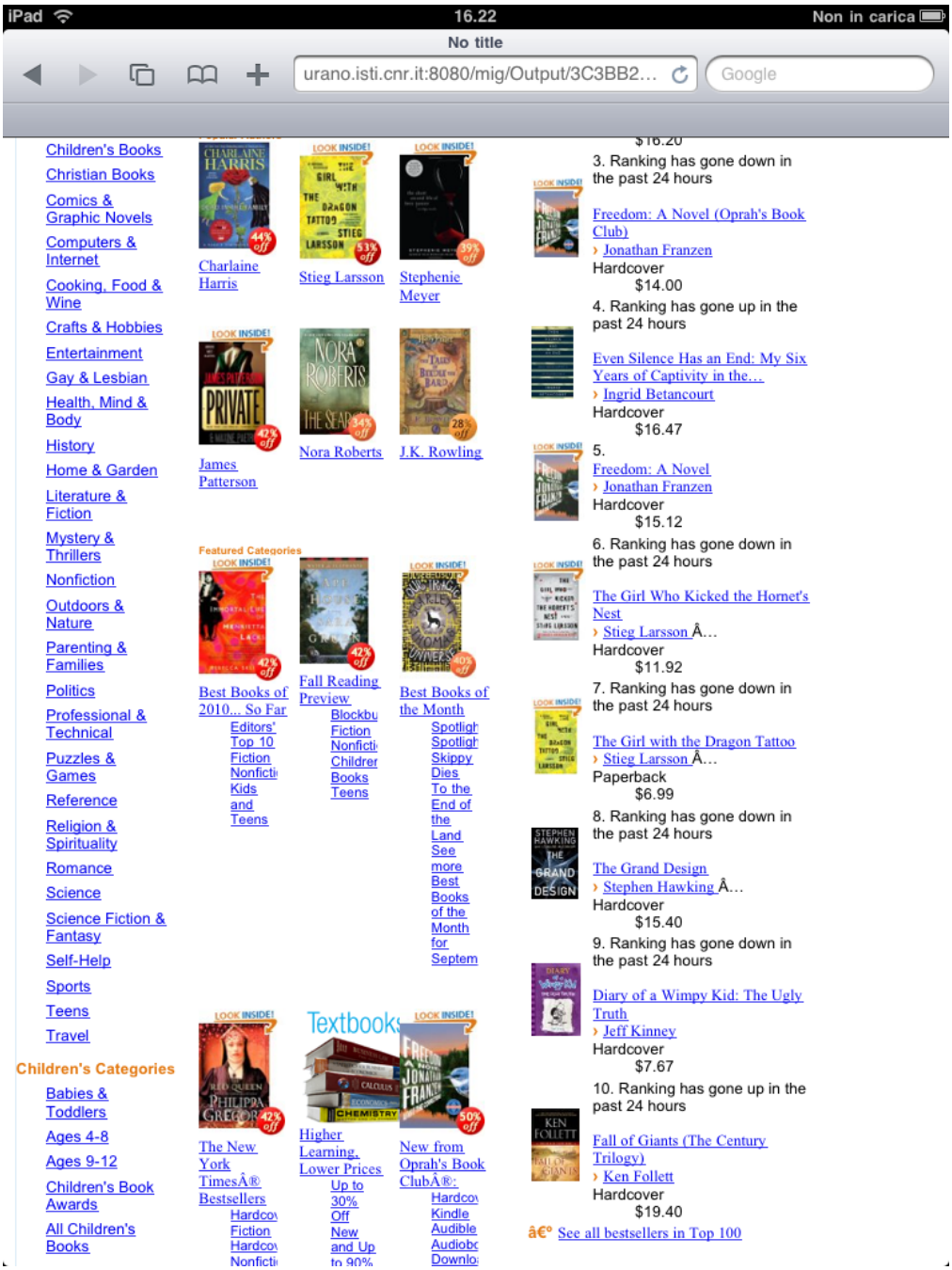

Figure 13: Giuseppe, iPad device, Amazon application totally migrated

3. CONCLUSIONS

This deliverable describes the final state of the prototype supporting the migration of the user interface software in Web applications. In particular, we provided an example of multiuser application on which both total and partial migration were carried out. The results obtained have shown a good potential of the solution, which works with various browsers.

Future work will be dedicated to the improvements of the parts dedicated to preserving the state of the JavaScript code, to perform optimizations in order to make more efficient the information flow across the various modules, and to extend the approach when secure http protocols are used.

4. BIBLIOGRAPHY

1. **OPEN Deliverable.** *D2.1: Early infrastructure for migratory interfaces.* 2009.

2. **OPEN Deliverable.**. *D2.5: Engineered infrastructure for migratory interfaces.* 2009.

3. Dojox.json.ref library, available at http://docs.dojocampus.org/dojox/json/ref