Project no.:
AAL-2009-2-137

PeerAssist

A P2P platform supporting virtual communities to assist independent living of senior citizens

Deliverable 5.1
"User interface implementation"

Lead Participant/Editor	Warp / Rafael Ramos
Authors	Rafael Ramos, Niki Rovatsou

Table of Contents

1 Introduction

In this deliverable we present the implementation of the User Interface subsystem. This work corresponds to the Task 5.1 - User interface implementation. Here we apply the technical design described in D4.2 to make the actual software component, which is integrated with the global architecture proposed in D4.1.

The first section reviews the implementation approach for the overall architecture of the PAnode, and states how the UIA is integrated in it.

The second section focuses on the internal implementation of the UIA module. It describes its structure, components, protocols and all the related technical aspects.

2 PAnode

The UIA is one of the agents in the PAnode, alonside PA, SLA, CA and HAC. All these are OSGi bundles in a container, and they interact with each other through their public interfaces.

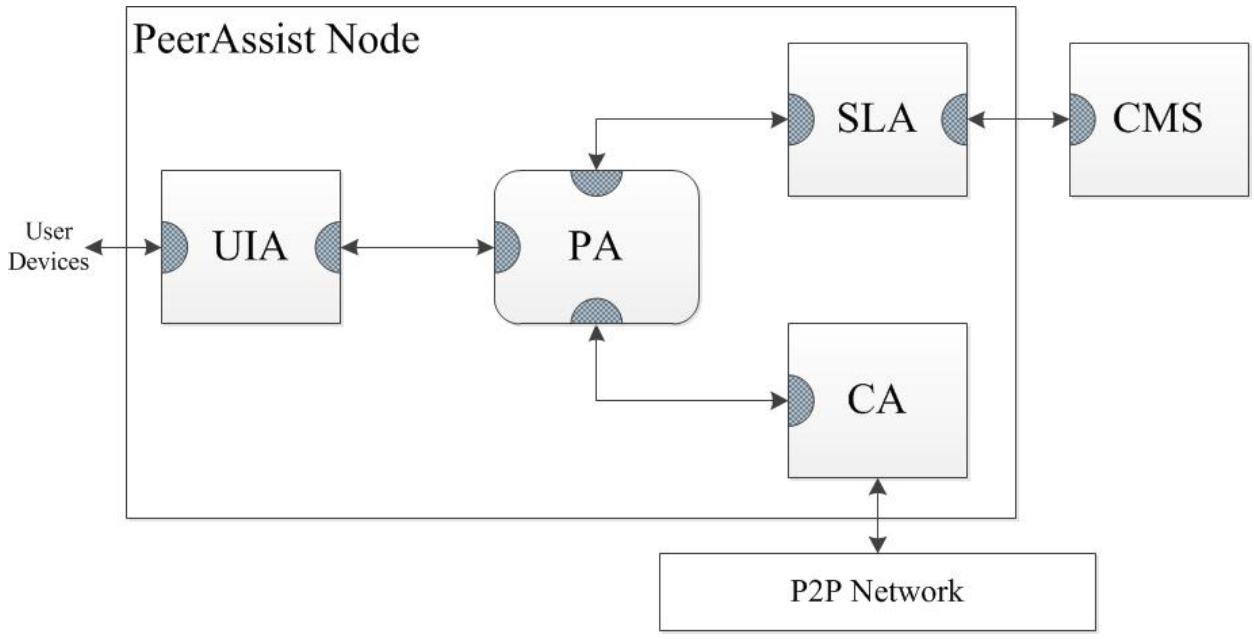

Figure 1: PAnode architecture

UIA requests backend functionlity through PA, which relays the method calls to SLA and CA. The defined interfaces are given in D4.1 annex. Most of the actions involve getting or sending data to the semantic repository (CMS), and performing operations in the P2P network.

As discussed in D4.2, the UIA is designed to support a range of user devices, including PCs, tablets, TV or even audio-only devices. This is enabled by the web-based design and the multimodal interaction approach.

3 UIA

The UIA agent is a rather complex system with several internal components and behaviours. The following sections describe in detail all the relevant aspects of its implementation, including the high-level architecture, the involved patterns and protocols, and some other technical issues.

3.1 Architecture

The high-level design for the UIA architecture is described in detail in D4.2. As explained, it is a GWT web application wrapped as an OSGi bundle. This allows to interact with other agents, as well as provide a web-based interface that includes a client-side component.

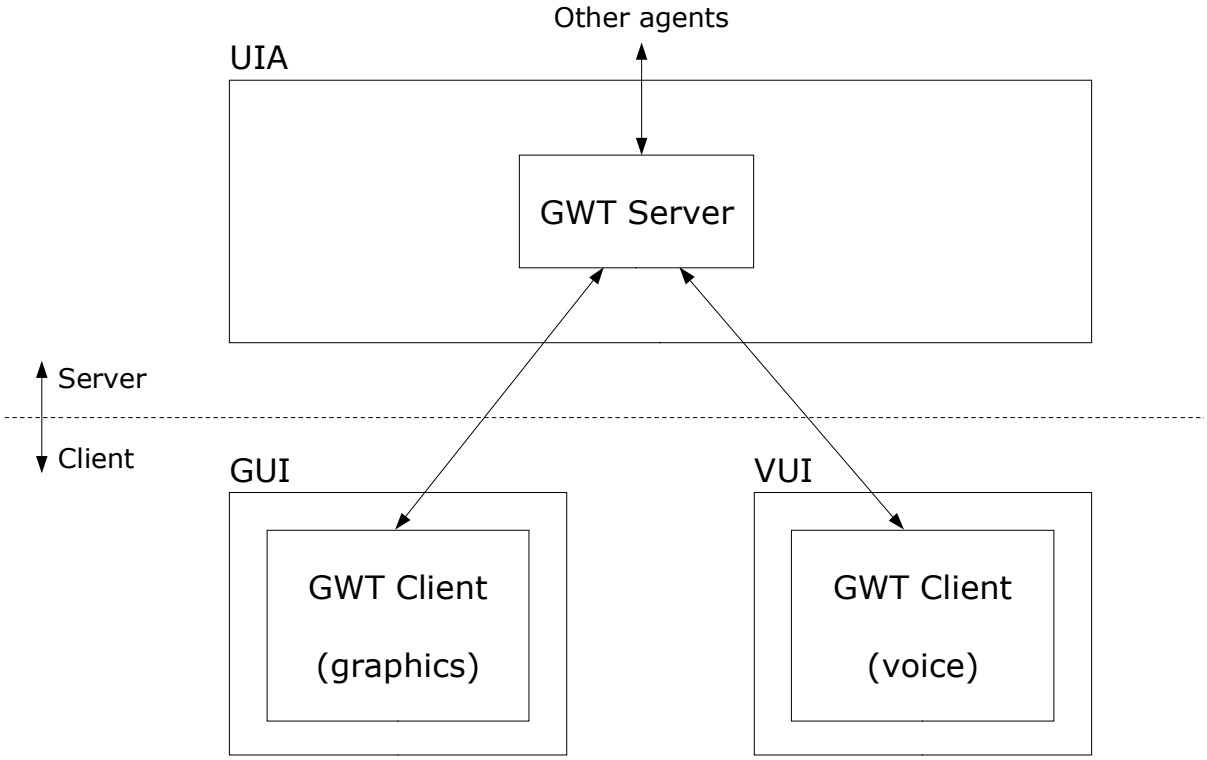

Figure 2: Multimodal architecture

The Multimodal architecture is achieved by allowing several instances of the client, of different modes, to interact with the server in a synchronized way. The clients may be in several diverse devices, such as PCs, tablets or smartphones. This design requires to implement the following key features:

- A GWT client component that offers two alternative modes, graphics and voice.
- An HTTP-based messaging protocol to keep synchronized the state of the clients with that of the server.
- A bi-directional interface to interact with PA.

3.2 Implementation

The concrete implementation of the UIA consists of a set of Java object components. The organization of these objects is based on the GWT project structure, with additional architectural features. The following diagram shows where these components are and how they are connected.

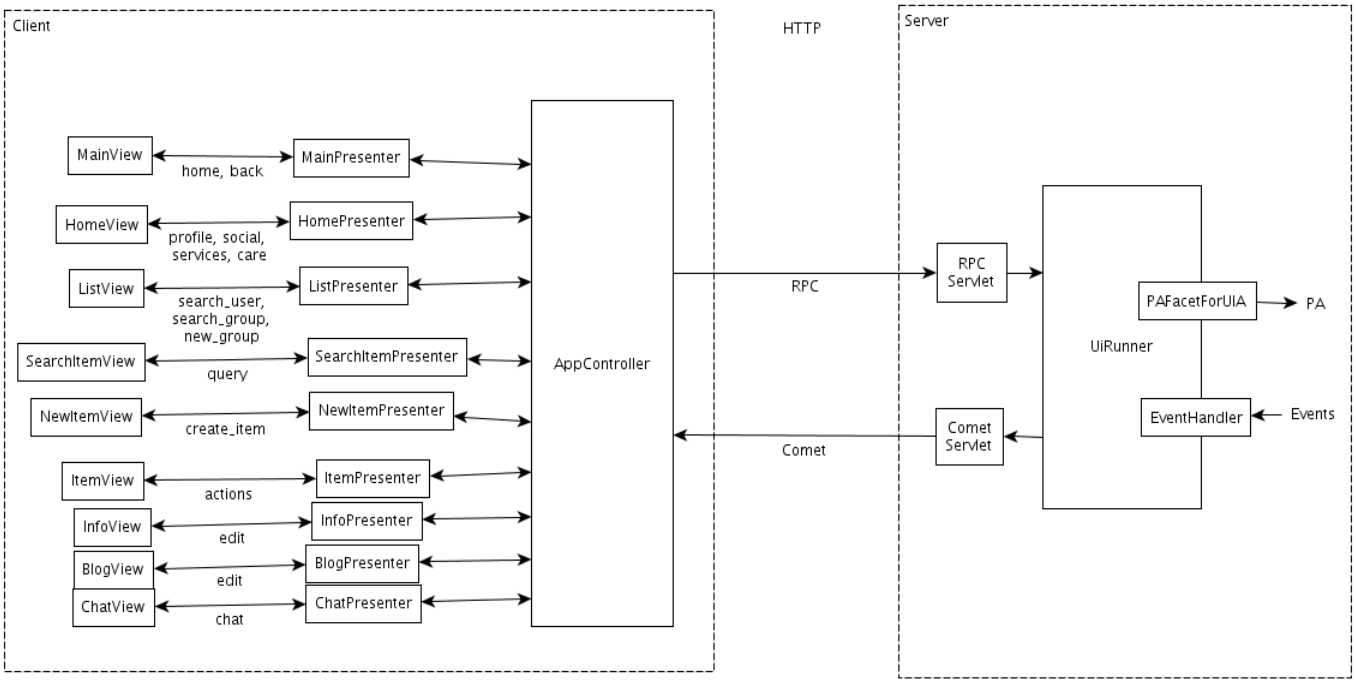

Figure 3: UIA implementation

The server side runs in the PAnode within the OSGi container. The client side runs as JavaScript in a web browser in the user's device. There may be several client instances.

The following sections describe in detail the relevant aspects of this implementation.

3.2.1 OSGi integration

The *UiRunner* is the main component in the server. It receives events from the client and executes the logic to perform the use cases initiated by the user. To do that it must communicate with PA.

PAFacetForUIA is the interface that defines all the PA methods exposed to UIA. The PA bundle has *PAFacetForUIAImpl*, which implements this interface. On bootup, the UIA Activator gets a

reference to the PA bundle (which should be loaded). From then on, *UiRunner* uses that reference to invoke methods on PA.

These calls are synchronous, i.e. the caller is blocked until a response arrives. This doesn't affect the user, who can keep using the client-side UI meanwhile. The calls may also return an Exception, which would be catched and handled by UiRunner.

On the other hand, some events may occur that originate in the backend, such as receiving network messages or triggering sensor signals. These must be passed to PA and UIA. To do this, the system makes use of the EventAdmin mechanism of OSGi: some bundles can throw Events with attached data and other can catch them. In this case the UIA registers itself as a Listener of Events of the following specific types:

- "peerassist/pa/userChatMessage/*"
- "peerassist/pa/groupChatMessage/*"
- "peerassist/pa/userVideoCall/*"
- "peerassist/pa/notification/*"
- "peerassist/pa/statusMessage/*"
- "peerassist/pa/Checkout/*"

The PA will throw events with these signatures, attaching the relevant data (e.g. a chat message sender and text). Then the UIA will receive them and UiRunner will have its handler method called, where it performs the necessary actions (e.g. display a chat message on screen).

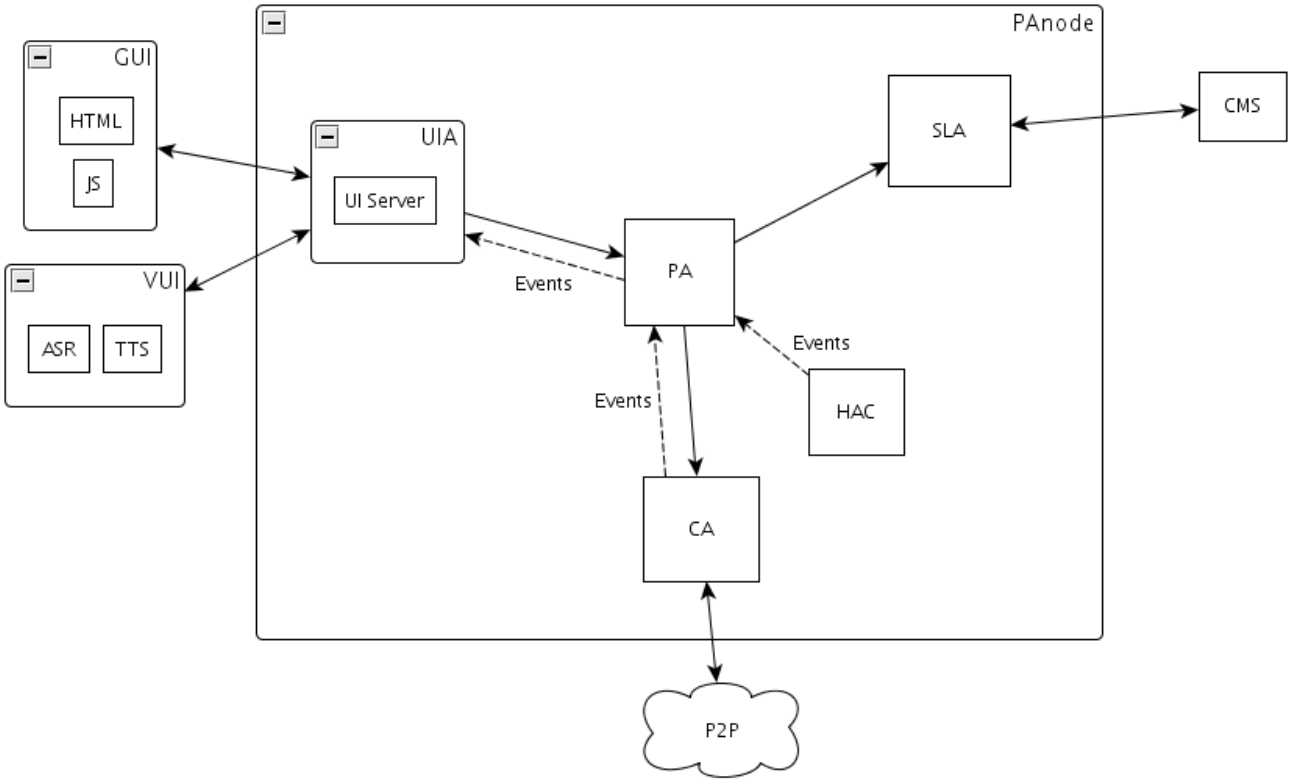

Figure 4: Communication between agents

As a rule, all UI-initiated actions are executed through method invocations towards PA, SLA and CA. Inversely, backend-initiated actions are passed towards PA and UIA through OSGi events. This approach eliminates cyclic dependencies between bundles.

3.2.2 Client-Server communication

As exposed in D4.2, this web application doesn't use the typical HTTP request-response model for communication. The multimodal strategy requires that any action initiated by one client is replied with responses to all clients, not just the originator. That way the user can operate whatever modality they prefer, and feedback will occur on all of the active devices.

This communication model requires two mechanisms. First, sending messages to the server asynchronously, i.e. not blocking the client waiting for a response. Second, delivering server-initiated messages to clients not in response to HTTP requests. These messages will carry user actions and data.

For client-originated messages, we use the GWT RPC (Remote Procedure Call) mechanism. This framework allows the client to invoke methods in the server transparently. When the client Java code is compiled into JavaScript, these calls are be actually performed as Ajax calls. The method is defined in an interface that is implemented in the server. The client must call an asynchronous

version of that interface so it doesn't block, instead the passed callback will be executed when a response arrives.

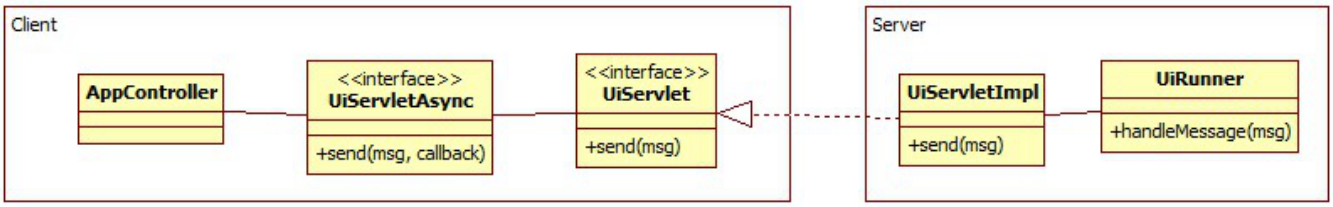

Figure 5: RPC communication

In the UIA approach, the *AppController* just uses a *send()* method to send a *UiMessage* to the server. That message includes a type and payload data, which *UiRunner* can handle. The response is ignored by passing an empty callback, since the client will receive feedback by a Comet message.

For the delivery of messages to all clientes, we use Comet communication. This technique allows to send server-initiated messages. We use the gwt-comet library, which implements that with "hanging GETs". The client permanently sends requests which are kept unresponded in the server until some event must be delivered. Then the HTTP response is sent, the client handles it, and then it makes another hanging request.

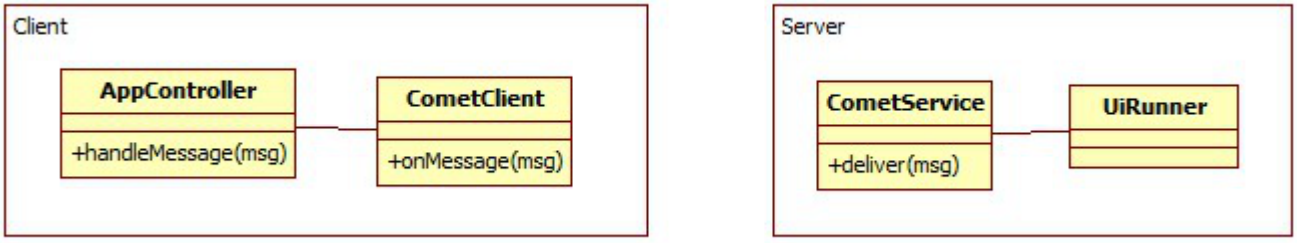

Figure 6: Comet communication

In the UIA implementation, *UiRunner* uses the *deliver()* method of a *CometService* that will transmit it to all clients. The client's *CometClient* will have its *onMessage()* method called, and *AppController* will handle it.

Regarding the actual messages, they must enclose the necessary information about user interaction. The *UiMessage* class has a type that indicates the kind of action to be performed, and some arbitrary payload data that may be needed. These are some of the most important message formats in each direction.

- **From client to server**
  - GO <place>: the user wants to navigate to some place.
    e.g. GO home
  - ACTION <action> <itemId>: request some action on an item.
    e.g. ACTION requestFriend john, ACTION deleteGroup g1....
  - QUERY <queryType> <queryParams>: execute a query of some type with the given parameters.
    e.g. QUERY searchUser {country: Spain}.
  - CREATE <itemType> <itemValues>: request creation of a new item of a given type, passing its data fields.
    e.g. CREATE group {name: "Poker Club"...}.
- **From server to client**
  - GO <place>: navigate to some place.
    e.g. GO home
  - SHOW <data>: display the data in the UI (e.g. a user's profile, the frieds list, the notifications...).
    e.g. SHOW {id: john, givenName: John, familyName: Doe...}
  - CHAT <type> <sender> <message>: display a received chat message.
    e.g. CHAT user john "hello!"
  - ERROR <errorMessage>: display an error message.
    e.g. ERROR "User does not exist"

The messages towards the server are actions the user does in the UI. The messages towards the clients are UI updates that must be rendered in all modalities. The complete list of message types can be seen in the source code of the *UiMessage* class.

The following example shows the message exchange between the two clients and the server when navigating to a user's page. The user requests from the GUI to go to John's profile. The server gives both clients the order to navigate to that page. Next, the GUI automatically requests to load all of John's data, so the server gives sends it to them (profile, blog posts and chat history). All UI actions follow a similar pattern.

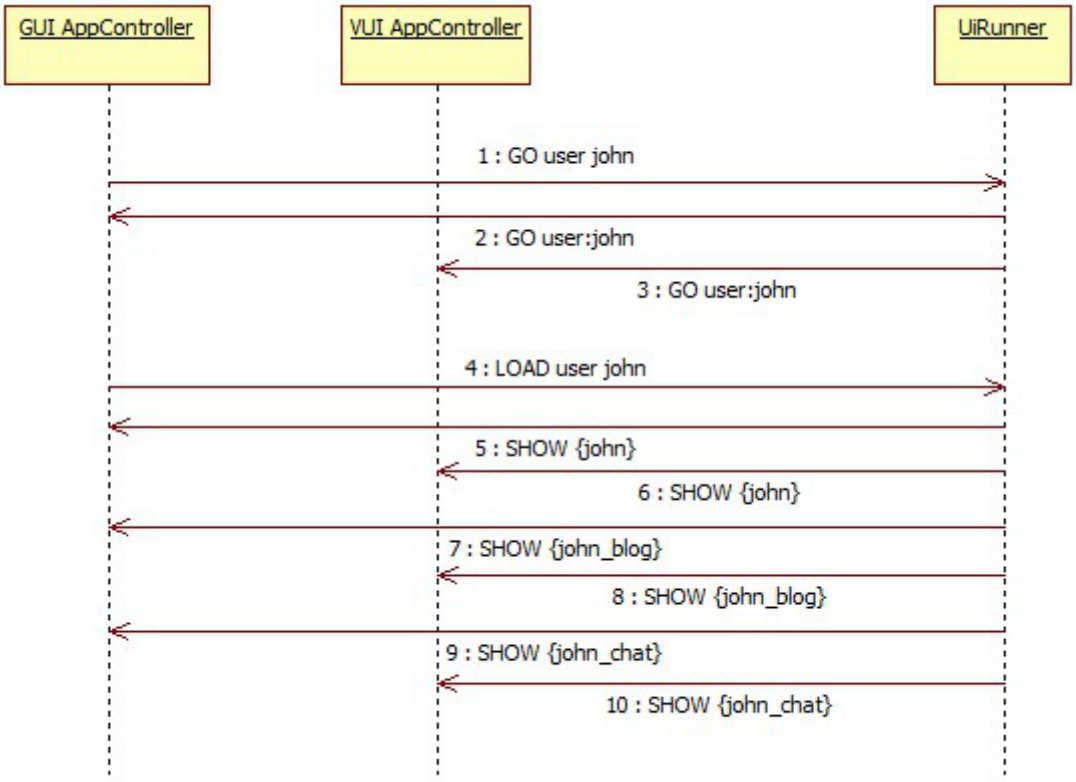

Figure 7: Client-server message exchange to navigate to a user page

3.2.3 Client MVP pattern

MVP (Model-View-Presenter) is a design pattern that aims at decoupling the components by organizing them in well-defined, separated layers. In D4.2 we discuss this pattern, and how we apply it to the GWT client. As mentioned, the main feature is the ability to have several different View implementations managed by shared Presenters through common interfaces.

The UIA client implementation is composed by the following components (see Figure 3):

- **AppController**: It's the most important object. It's in charge of communicating with server through the RPC and Comet protocols described above. On the other side, it controls all the Presenters, manages their lifecycles and gives them data.

- **Models**: These are just the data objects that are passed around. Some of them are *UiUser*, *UiGroup*, *UiService*, *UiNotification*, *UiPost*, *UiChatMessage*, etc.

- **Presenters**: These are the components that drive the Views. There is one for each View type, e.g. Home, List, Item. the Presenters keep the state of the Views (e.g. current item), they are created when arriving to the page, and destroyed when leaving. Their methods are invoked by Views when any user action occurs. Also, they receive data from AppController and push it to UI.

- **Views**: These components implement the actual presentation layer. There is one for each page type, and they are linked to their corresponding Presenter. For optimization, they are reused and reset as the user navigates again to the pages. The Views are defined by interfaces.

  The View interfaces define a set of methods based on the data they must show (e.g. an item, a list, a form). The different View implementations offer this same interface, but they execute the methods depending on its mode, i.e. graphics or voice. Graphic Views will render visual widgets, whereas Voice Views generate spoken messages. The Presenter doesn't need to know what type the View is.

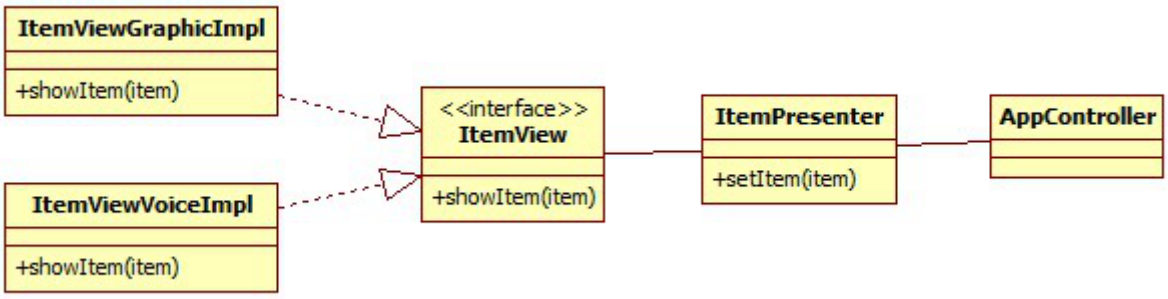

Figure 8: MVP pattern for Item View

The GWT client includes both versions of the Views. The user can select the modality through a selector or with a URL parameter (e.g. http://localhost:8080/uia/Uia.html?mode=vui). Upon initialization, an abstract *ViewFactory* will be instantiated in either mode, and it will be used to make the concrete View instances, either graphics or voice versions.

There are several Views, with their associated Presenters and implementations, coresponding to each of the page templates involved in the UI:

- **MainView**: The Main container for the UI, displays the header and footer, and contains other views.
- **HomeView**: The Home page, displays navigation buttons.
- **ListView**: The Social, Services and Care pages, displays lists of items.
- **ItemView**: The User, Group and Service pages, displays the item, action buttons, and includes sub-views as tabs:
  - **InfoView**: The item information, may be editable.
  - **BlogView**: The User or Group blog, the posts may be editable.
  - **ChatView**: The User or Group chat.
  - **VideoView**: The panel to do video calls with the User.
- **SearchItemView**: The SearchUser, SearchGroup and SearchService pages, displays the query form and results list.

- **NewItemView**: The NewGroup and NewService pages, displays the item form and suggestions list.
- **AdminView**: The Admin page, displays an administration panel.
- **LoginView**: The Login page, where the user enters username and password.

Some views can be nested inside others, for example the sub-views of ItemView. The MainView is always visible and contains the others.

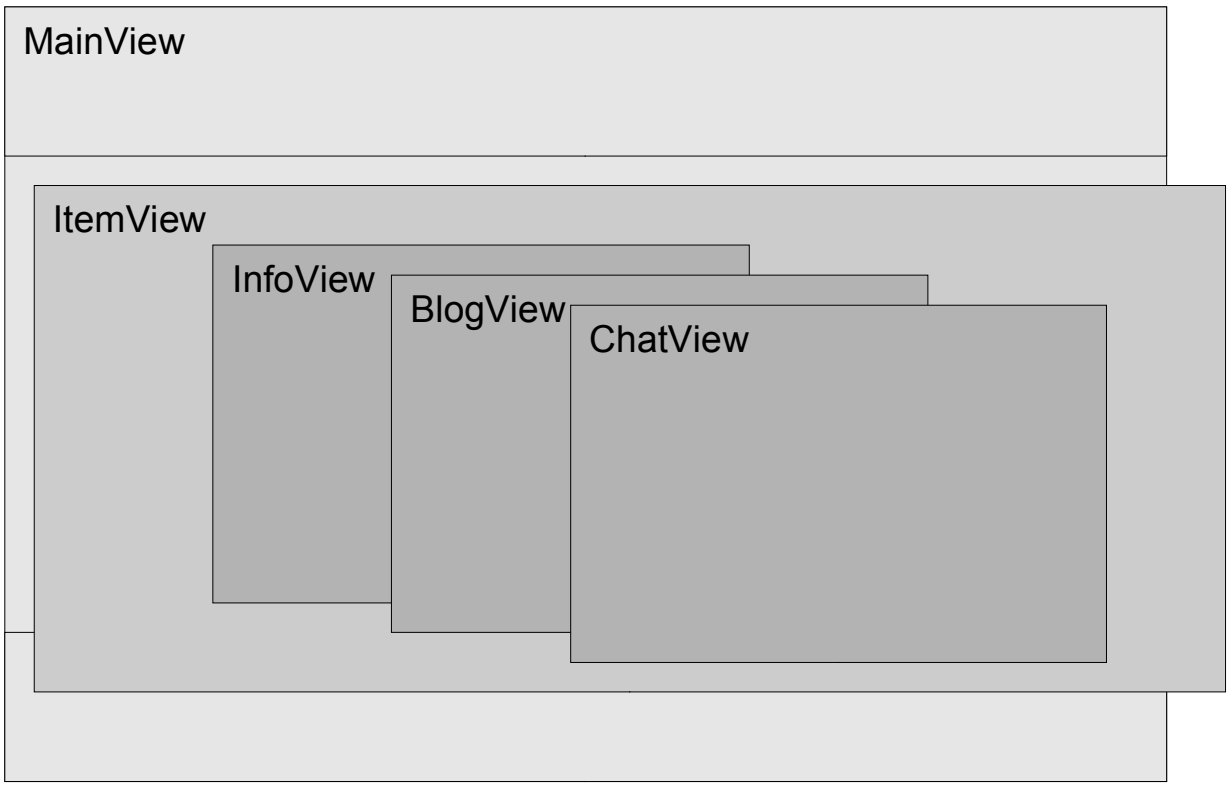

Figure 9: View nesting

3.2.4 Navigation

The UIA is designed as a single-page application. This is, only one page is loaded in the browser at startup, and all subsequent activity happens inside this "host page" controlled by the JavaScript client-side component. This is in contrast to regular web apps, where the navigation is done by HTTP requests and responses.

As described, all interaction with the server is handled by the Multimodal protocol. This includes sending and receiving data, as well as navigating throughout the pages.

The app starts by displaying the MainView, consisting of the header, footer and a central panel. Then, it will render all pages in that central container, each time removing a View to load the next one. The first one is the Home page if the user is logged, or the Login page otherwise.

Navigation is done by sending GO messages to the server, which responds with GO commands to all clients so all of them advance to the target page. Also, a "GO back" message can be sent, which cause the browsers to go back one page in their History, thus reloading the previous View within MainView.

3.2.5 Graphics

The GWT framework provides a toolkit to build graphical user interfaces. There is a set of available widgets, that can be created and configured according to the intended behaviour. In some cases, these widgets are instantiated programatically from the Java code in the View implementations.

However, there's an alternative mechanism called UiBinder. This tool allows to declare UI components in XML, which usually results in easier programming and more compact code. These templates allow to include HTML and CSS code, which gives high expressiveness and flexibility to customize the appearance and behaviour.

Most of the UIA Views are defined with UiBinder. The *XViewGraphicImpl.java* classes are usually linked to a *XViewGraphicImpl.ui.xml* file. The XML defines the layout and widgets, while the class specifies their behaviour. The most used widgets are Labels, Buttons, TextAreas, Lists and Images.

3.2.6 Voice

In Voice mode the UIA dynamics are the same as in Graphics mode, except that it uses the voice versions of the Views. The GWT client is still a JavaScript app that manipulates the DOM within an HTML host page. However, these Views manage some speech-based elements rather than graphical widgets.

As with graphics, the Main View is always active and it contains other Views as the user navigates. *MainViewVoiceImpl* is responsible for handling speech input and output from its JavaScript code, which makes use of the Google Chrome speech API.

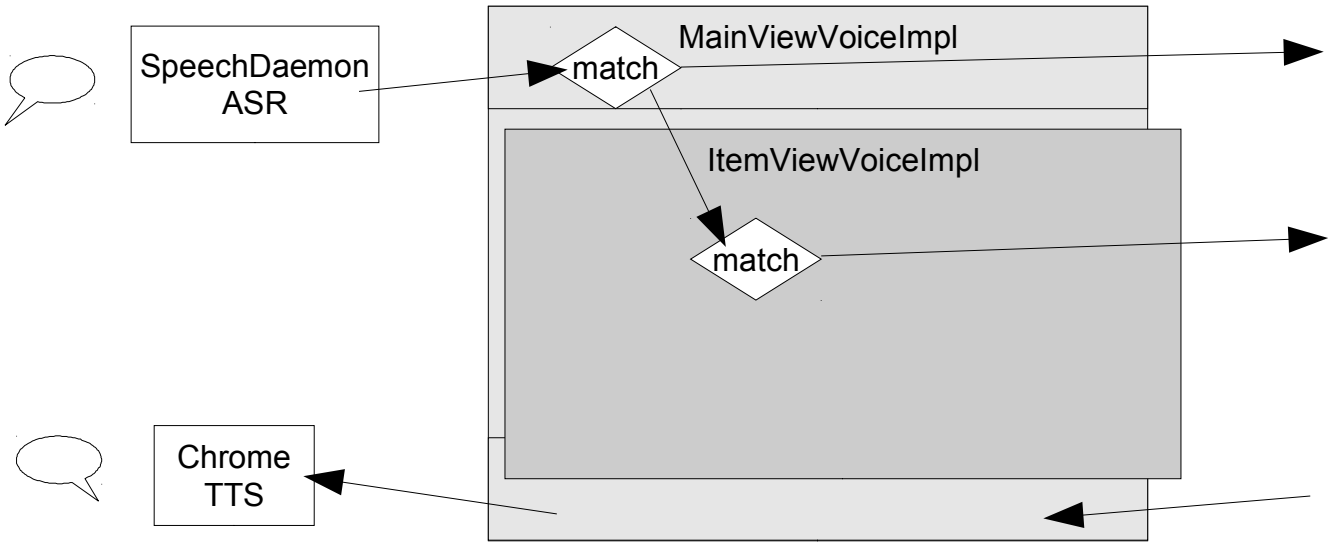

Figure 10: Dynamics of Voice interface

For speech recognition we developed a Chrome extension named *SpeechDaemon* which permanently listens for sound from the microphone, in a loop. It feeds this audio stream to the built-in Chrome ASR system, which actually uses a Google web service to perform the recognition. Whenever a message is recognized, the daemon receives it as a string and passes it to *MainViewVoiceImpl*.

The MainView tries to match the string against the globally accepted commands, i.e. those that are always available. These are "start", "stop", "back", "home", "profile", "social", "services", "care", "notifications", and "alarm". If the string matches one of them, the action is invoked in the server, as if the user clicked that button. Else, the string is passed to the contained View being active at that moment (the current page). This View (e.g. *ItemViewVoiceImpl*) will try to match it against its particular available commands, which are similar to those in their graphic counterpart (e.g. for a user page they would be "info", "blog", "chat", "friend", "unfriend", "invite", etc.). Again, if a match is found the action is requested to the server; else, the command is ignored.

Regarding the output, *MainViewVoiceImpl* will simply pass the received message string to the Chrome TTS module, which reads it aloud. This is similar to the Graphic version, where the message is displayed in the assistant text area.

3.2.7 Video

The video call functionality of the PeerAssist platform is based on the OpenTok open source platform, and the service the platform provides. The OpenTok API covers a wide variety of platforms, as it is available for Java, Python, iOS, .Net, Ruby and PHP. The client-side implementation, is available for Javascript and Actionscript, with the first one being the best option. The latest platform improvements, offer an HTML5 implementation, which is much faster and easier to use in an external application, given that there will soon be a stable version.

When a user initiates a video call with another user, the following things happen:

1. The server site OpenTok library, generates a session:

   OpenTokSDK → create_session('127.0.0.1')

2. The server site OpenTok library, generates a token:

   OpenTokSDK → generate_token(sessionId)

3. A simple HTTP GET Request is sent to the OpenTok online service, including the session id and the token that where created in the previous steps.

4. A notification is sent to the other user, along with a simple message that includes the session id and the token.

5. If the user decides to answer the video call, another HTTP GET Request is sent to the OpenTok service, including the same session id and token that were sent before.

6. The OpenTok service acts as an intermediate between the two users, transferring the video call from one to user to the other, taking care of the security of the call, by authenticating both users, using the session Id (like a “private room” id) and the token (like a password) provided. After the authentication, the users are connected in a peer to peer manner, without the need of an intermediate between them to stream the video and audio from one user to the other.

7. When one of the users wants to end the video call, a *disconnect()* method is executed on the session object, terminating the video streaming between the PeerAssist users.

The video call implementation uses the Flash capabilities of the browser, which means that the users' browsers must have Flash enabled, and set using the instructions provided in the PeerAssist installation.

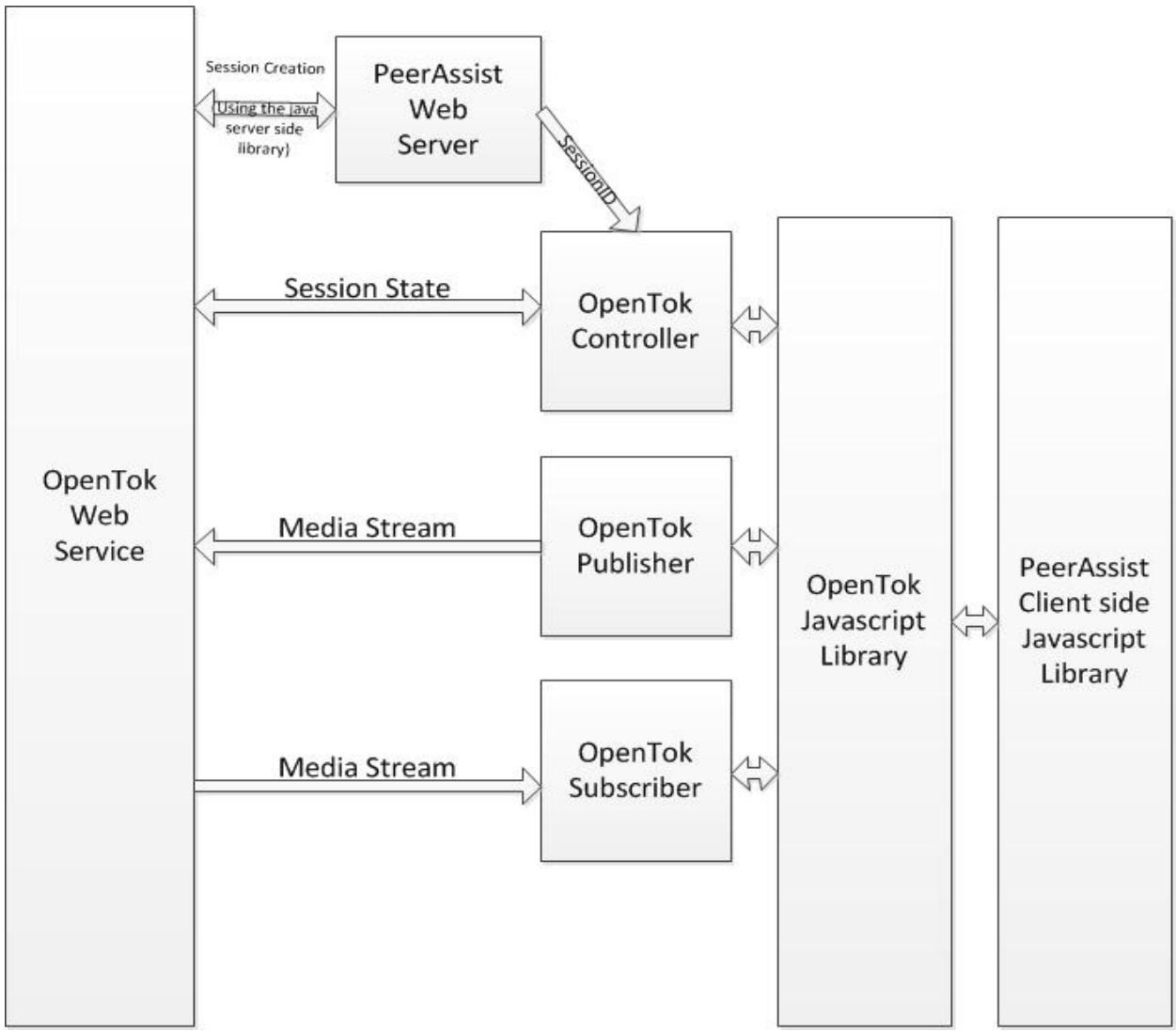

Figure 11: Video call implementation

The server side library is the *com.opentok.api* library, including the OpenTokSDK (the main OpenTok class including all the functionality), the OpentokSession (the class representing the session object created) and the TokBoxNetConnection (the class responsible for creating the connection between the client application and the OpenTok server).

3.2.8 i18n

The UIA implements internationalization to support multiple languages. Currently there are English, Spanish and Greek translations available.

GWT includes a flexible set of tools to help you internationalize web applications and libraries. GWT internationalization support provides a variety of techniques to internationalize strings, typed values, and classes.

The GWT internationalization types reside in the com.google.gwt.i18n package. To use any of these types, the module must inherit from the I18N module (com.google.gwt.i18n.I18N).

In Peerassist we used Static String Internationalization. Static string internationalization is the most efficient way to localize web application for different locales in terms of runtime performance. This approach is called "static" because it refers to creating tags that are matched up with human readable strings at compile time. At compile time, mappings between tags and strings are created for all languages defined in the module. The module startup sequence maps the appropriate implementation based on the locale setting using deferred binding.

Static string localization relies on code generation from standard Java properties files or annotations in the Java source. GWT supports static string localization through three tag interfaces (that is, interfaces having no methods that represent a functionality contract) and a code generation library to generate implementations of those interfaces.

Standard Java properties files are placed into the same package as the main module class. They must be placed in the same package as their corresponding Constants/Messages sub-interface definition file.

All Peerassist internationalization file are placed in the client/locales folder of the main module class (es.warp.peerassist.uia).

An example of such an internationalization file from the Peerassist web application is presented below:

```
package es.warp.peerassist.uia.client.locales;

import com.google.gwt.i18n.client.Messages;

public interface UiaClientConstants extends Messages {
   String home();
   String back();
   String profile();
   String social();

}
```

```
home=Inicio
back=AtrÃ¡s
profile=Perfil
social=Social

infoTabTitle=InformaciÃ³n
blogTabTitle=Blog
chatTabTitle=Chat
webTabTitle=Web
videoTabTitle=VÃdeo

profileInfoWelcome=Este es tu perfil, haz click en editar para cambiarlo.
profileBlogWelcome=Este es tu blog.
```

3.2.9 Login

The UIA uses an authentication mechanism to protect from unauthorized access and ensure privacy. At startup the user must login with their user and password. If the credentials are valid, a session cookie is generated in the server and passed to the client. All subsequent HTTP requests will carry that cookie, and the server will check if it's valid in order to authorize the requested action.

When the user logs out the session cookie is destroyed. The user must login on all the browsers or devices they intend to use, e.g. for graphics and voice combined interaction.

3.2.10 Styles

The UIA uses Cascading Style Sheets (CSS) to define the visual appearance. CSS is a style sheet language used for describing the presentation semantics (the look and formatting) of a document written in a markup language. Its most common application is to style web pages written in HTML.

CSS is designed primarily to enable the separation of document content (written in HTML or a similar markup language) from document presentation, including elements such as the layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple pages to share formatting, and reduce complexity and repetition in the structural content.

An example of CSS coding is presented below:

```
h1 {

 font-size: 2em;
 font-weight: bold;
 color: #777777;
 margin: 40px 0px 70px;
 text-align: center;
}

.sendButton {
 display: block;
 font-size: 16pt;
}

/** Most GWT widgets already have a style name defined */
.gwt-DialogBox {
 width: 400px;
}

.dialogVPanel {
 margin: 5px;
}
```

```
.serverResponseLabelError {
 color: red;
}

#closeButton {
 margin: 15px 6px 6px;
}

.profileColor {
 background-color: #fed298;
}
```

4 Updated version of GUI

Following the users' comments we implemented a complete redesign of the Peerassist GUI in order to be more user friendly and more attractive in terms of its graphics. For that reason, images and icons are designed from scratch and new colors are selected so as to provide a more elegant version on user screen.

The new design uses a simple interface pattern and highly relevant graphics and pictures that are more significant than the detailed decorations of the old GUI version.

In the remaining of this section we present a number of indicative screenshots of the new version of GUI in comparison with the initial version.

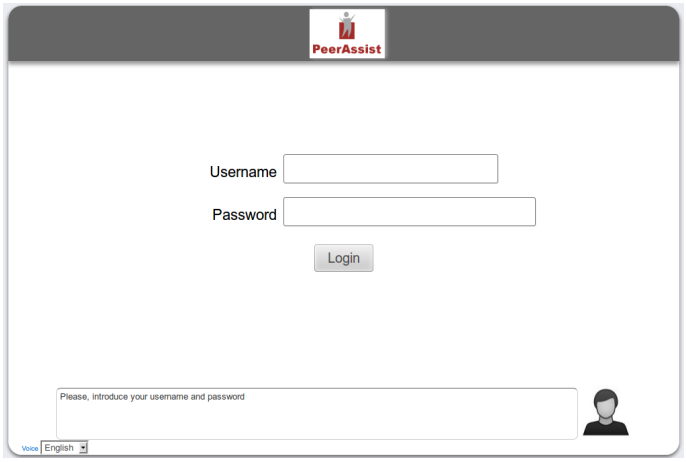

Figure 12: Login page - Initial version

Figure 13: Login page - Updated version

Figure 14: Home page - Initial version

Figure 15: Home page - Updated version

Figure 16: Profile page - Initial version

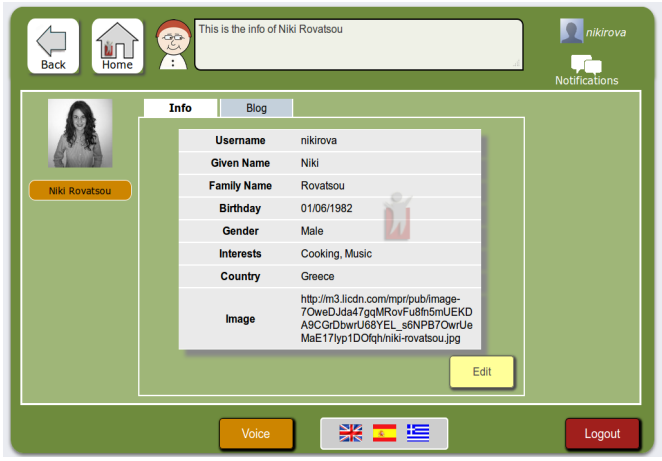

Figure 17: Profile page - Updated version

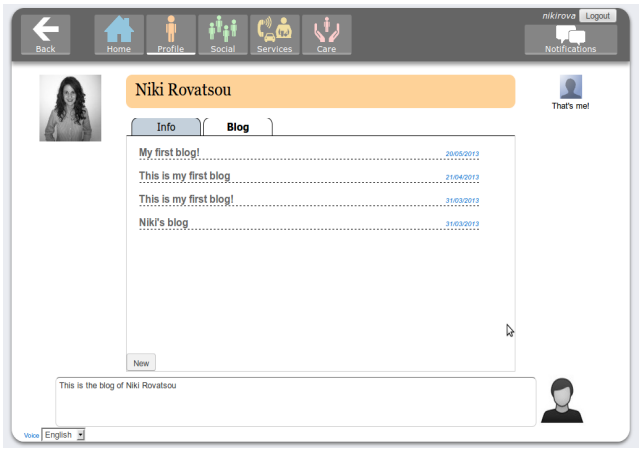

Figure 18: Profile page/Blog - Initial version

Figure 19: Profile page/Blog - Updated version

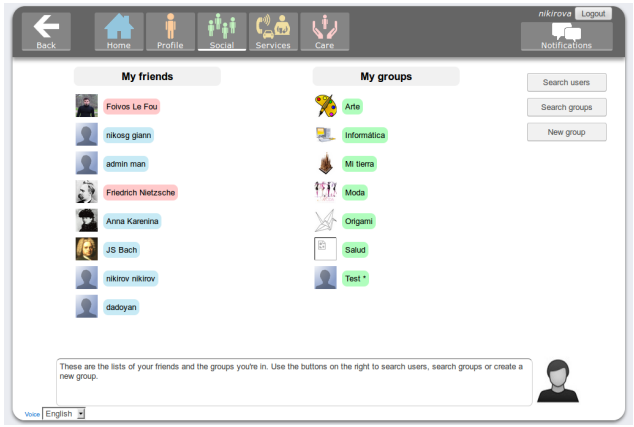

Figure 20: Social page - Initial version

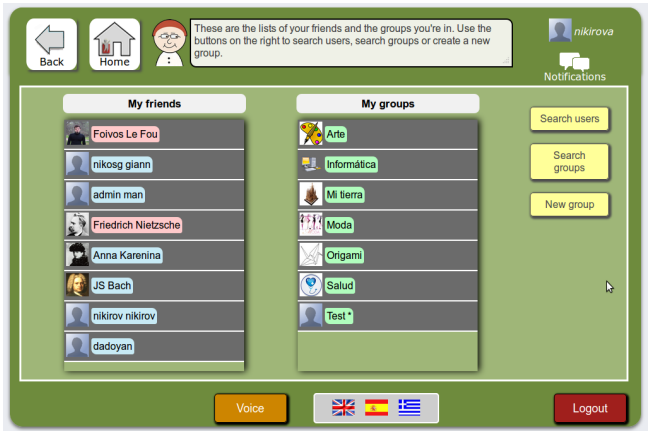

Figure 21: Social page - Updated version

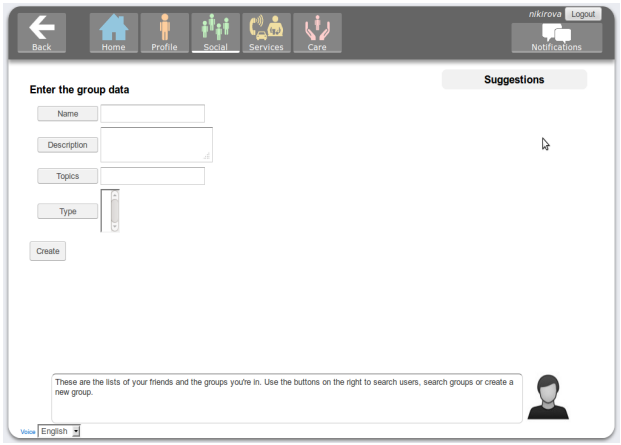

Figure 22: Social page/Create group - Initial version

Figure 23: Social page/Create group - Updated version

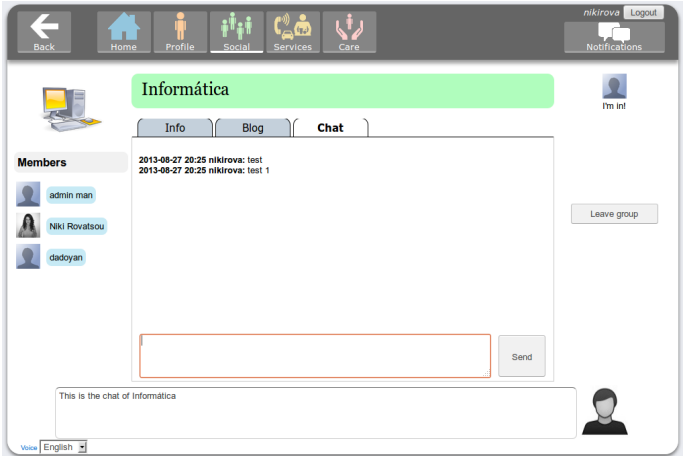

Figure 24: Social page/Chat - Initial version

Figure 25: Social page/Chat - Updated version

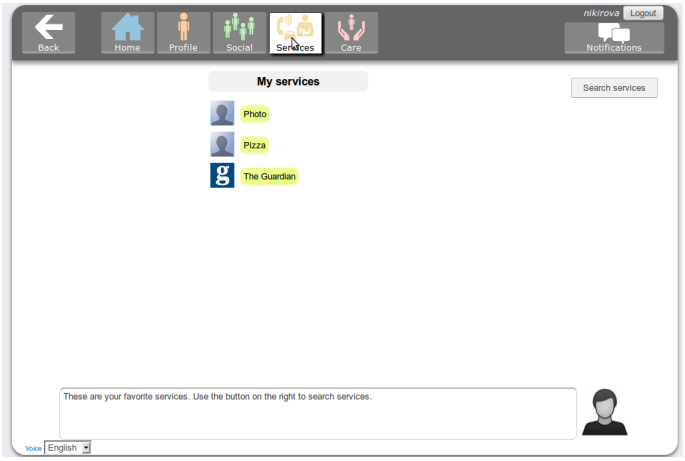

Figure 26: Services - Initial version

Figure 27: Services - Updated version

Figure 28: Search services/ Initial version

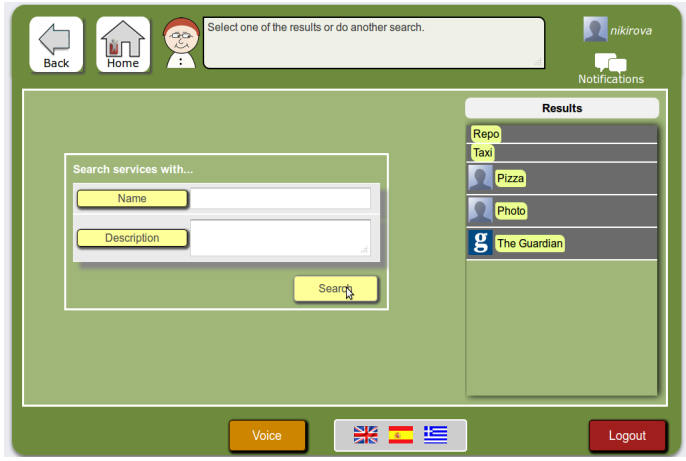

Figure 29: Search services/ Updated version

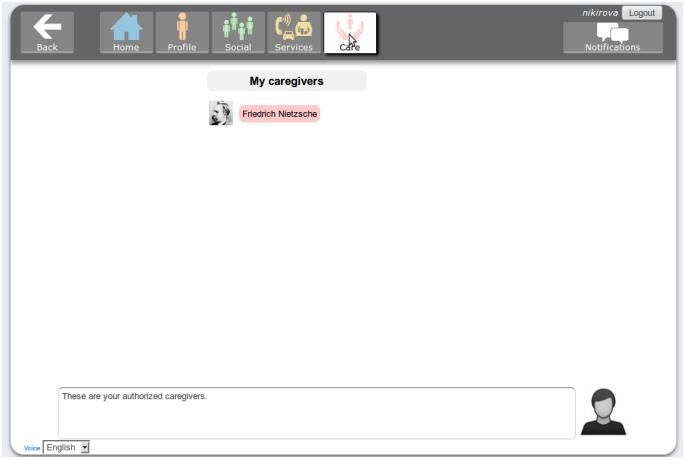

Figure 30: Care page/ Initial version

Figure 31: Care page/ Updated version

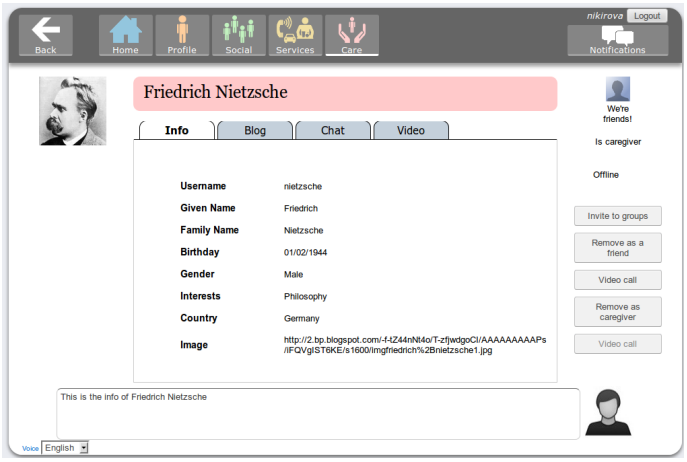

Figure 32: Caregiver profile/Initial version

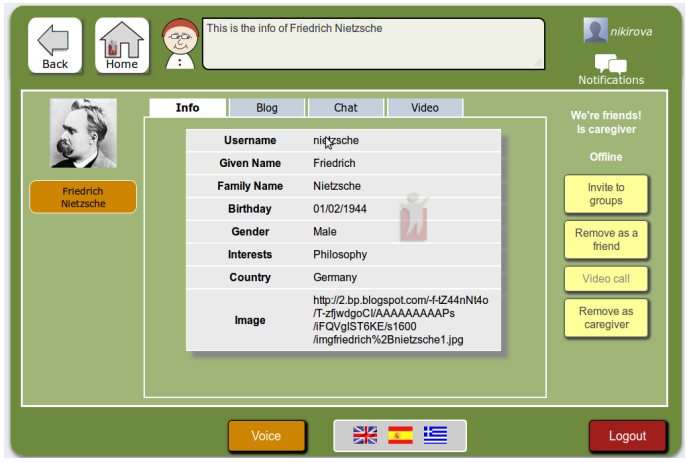

Figure 33: Caregiver profile/Updated profile

5 Conclusions

In this task we implemented the proposed design for the UI subsystem, turning it into an actual working software component.

We tried many of the available technologies to choose the most suitable for each case. During the development phase many decisions were made on technical aspects, sometimes involving substantial changes respect to the original design. However, we managed to adapt the approach to drive the development towards completion. Also, the work was carried on in close collaboration with the developers of the other modules, to ensure compatibility and coherent integration.

The resulting UIA is able to offer an accessible, muiltilingual, multimodal user interface that allows the user to perform the use cases in the PeerAssist platform. The successful completion of this functional UIA proved the design to be feasible and appropriate to fulfill the requirements.