Project no.:
AAL-2009-2-137

PeerAssist

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

Deliverable D4.1
"The overall system architecture and implementation platform"

Lead Participant/Editor	UoA / Elias Manolakos
Authors	Elias Manolakos, Rafael Ramos Guallar, Michael Fried, Elimia Cimpian, Christos Xenakis, Christos Anagnostopoulos

Table of Contents

1 Introduction 1
2 Top-level view of the software architecture 1
2.1 Conceptual basis of the PeerAssist platform 1
2.2 PeerAssist node 2
2.3 PeerAssist community networks 3
2.4 The PeerAssist matching and query system 4
3 The User Interface layer 5
3.1 Hardware deployment 5
3.2 Key solutions 6
3.3 Software components 7
3.3.1 Interaction Manager 7
3.3.2 UI Server 8
3.3.3 GUI Client 8
3.3.4 VUI Client 9
3.4 Component interfaces 9
4 The Semantic Layer 9
4.1 Semantic representation of PeerAssist concepts 9
4.2 Semantic Layer Architecture 10
4.3 Semantic Layer Components 11
5 The Communication Layer 11
5.1 Security functionality 12
5.2 Grouping functionality 13
5.3 Service functionality 13
6 The Personal Assistant Architecture 13
6.1 The Personal Assistant Module 14
6.2 The intelligence functionality of the PA 15
6.3 Dispatching functionality of the PA 18
7 The PeerAssist platform architecture 18
7.1 Platform components & design constraints 18
7.2 Home platform design and architecture 21
7.3 System architecture and integration 22
7.3.1 End user device 22
7.3.2 Voice platform 25

7.4 Examples of use cases mapped to the platform modules.....27
8 Conclusions 27
9 Appendix A – UML Activity Diagrams for the Use Cases.....28
9.1 Create Group.....28
9.2 Join Group and Leave Group.....30
9.3 Delete Group.....31
9.4 Invite Users.....32
9.5 Remove User from Group.....33
9.6 Search Item.....34
9.7 Advertise Event.....35
9.8 Raise Alarm.....36
9.9 Organize Event.....37
9.10 Consult Doctor.....37
10 Appendix B – Defined Interfaces.....39

1 Introduction

In this document, we provide an overview of the PeerAssist software and hardware system architecture, including all its layers, top-level components and their interactions. The individual top-level components are also analyzed to identify their major modules (subcomponents). The purpose of this deliverable is to clarify the components structure and role in the overall architecture, in order to provide functional specifications that will drive their implementation. This is accomplished by also analyzing all PeerAsssit use cases by empoying UML diagrams where each major component, its activities and how they interact in the use cases are identified.

2 Top-level view of the software architecture

2.1 Conceptual basis of the PeerAssist platform

Before proceeding with the design of the architecture, let us established a clear conceptual framework of PeerAssist that is summarized by the figure below

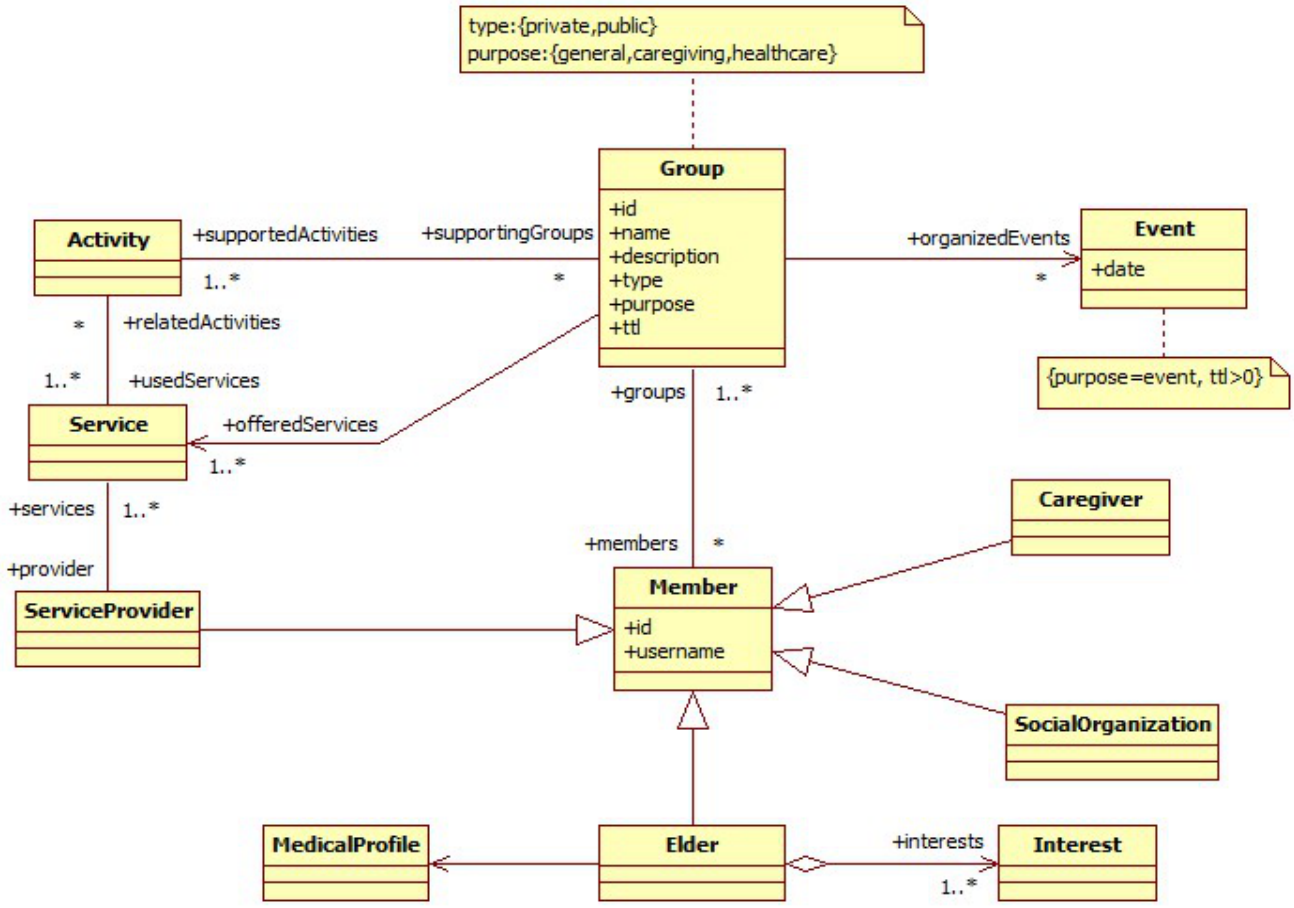

Figure 1: Conceptualization of the PeerAssist system

According to this conceptual model, a *Group* has a unique identifier, and is further characterized by a name, a description, a type (i.e., either private or public), a purpose (i.e., general, event, care-giving, or healthcare related),

and a time-to-live. A group may participate or organize *Events* (e.g., a social gathering) or support an *Activity* (e.g., an online discussion activity).

A Group is populated by at least one *Member* and, reversely, a Member may participate in one or more groups. Each member is distinguished by its identifier, while also having a user name. There are many specializations of the core Member concept, namely the *Caregiver*, the *Social Organization,* the *Service Provider,* and the *Elder*. The latter specifies one or more *Interests* and has a specific *Medical Profile*.

Each Group supports one or more *Activities* (e.g., a discussion about a specific topic, playing an online game, etc.), which are realized through a set of *Services*. It should be noted that, a particular Activity may be supported by more than one groups. Also, a particular service may be used in the context of more than one activities. For example, the chat service could be used both in the discussion and online game activities. Finally, services are provided by *Service Providers*, which are seen as a specialization of the Member concept.

Each group may organize one or more *Events* and an event has a purpose and a date associated with it.

2.2 PeerAssist node

The main software entity of PeerAssist is the *PeerAssist node* (PAnode). The PAnode is a complete software architecture running in each user's PC and interacting with the rest of the entities of the PeerAssist platform (i.e., the end-user device implementing the user interface, available sensors, etc.), in order to orchestrate the execution of use cases that provide the PeerAssist functionality. The top-level architecture of the PAnode is summarized in the figure below:

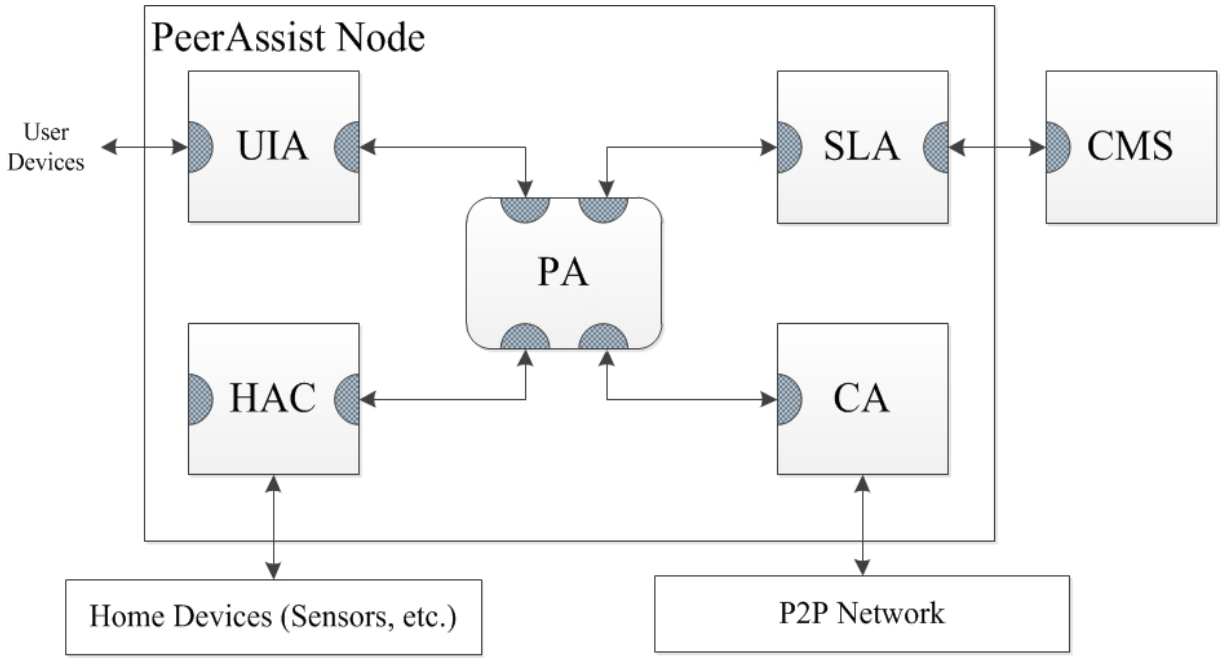

Figure 2: The PeerAssist node top-level software architecture

As it can be seen from the figure the major software components (to be called *agents*) of the PAnode are:

The ***User Interface Agent*** *(UIA)*. It is responsible to:

- Capture the end-user intent by interacting with the end-user device.

- Gather and communicate data that are used to form user queries.

The ***Semantic Layer Agent*** *(SLA)* that is responsible to:

- Maintain the user profile and local context information in the **Local Semantic Information** (LSI) knowledge base.
- Interact with the **Central Matching System** (CMS) when needed to process user queries. The CMS and its role will be discussed in section 4.

The ***Personal Assistant (PA)***, that is responsible to:

- Dispatch information between the rest of the agents acting as an intelligent intermediary
- Mediate whenever needed to facilitate interaction and support autonomy in certain use cases
- Gather and maintain relevant information to improve interaction with the user through machine learning

The ***Communication Agent*** *(CA)* that is responsible to:

- Handle all peer-to-peer (P2P) communications with other PAnodes in the PeerAssist network, thus abstracting the communication aspects of the design for the rest of the agents.
- Handles security and trust related issues.

The ***Home Automation Controller*** *(**H**AC)*, that is responsible to:

- Handle control of and communication with local user devices and sensors.

2.3 PeerAssist community networks

A typical use case in PeerAssist consists of a workflow with the following sequence of steps (not all steps are necessarily present in every use case):

1. *Capturing end-user intent:*
   This is accomplished using the UIA and is facilitated by taking into account the user's profile and context (available in the LSI). Potentially it may also involves the Personal Assistant (PA) in an assistive/suggestive role.

2. *Building a user community (group) to participate in an event/activity:*
   The Semantic layer (SLA-CMS) suggests which peers may join the group based on intent, constraints, user context, profile, etc. Existing groups or peers are presented to the user (through the UIA) and allowed to join the associated peer group (an ad hoc user community maintained by the CA). Users can join and leave a group dynamically on demand.

3. *Execution and progress monitoring of the use case:*
   It amounts to implementing the "business logic" of the use case application (task). A task may involve calling several services . Some services may be offered by local devices interacting with the

PeerAssist node, and some others may be global services with implementations obtained from a remote service repository.

Following this generic use case paradigm, a PeerAssist activity execution may necessitate the creation and maintainance of a logically fully connected graph of PAnodes. As shown in the figure below, a specific end-user (called the "originator") uses its PAnode (shaded node) to create a group (user community). Through interaction with the user interface a query is formed (step 0). The query may need to be sent all the way to the remote Central Matching System (CMS) in order to find matching users (step 1). The CMS returns a list of matching users rank ordered based on relevance (step 2). The originator-user filters the matches and sends invitations to a selected subset. Once these users accept the invitation a logical communication channel (pipe) is opened among pairs of peers participating in the formed community (step 3). These pipes (shown as red edges) are maintained by the CA for as long as they are needed to support the interaction of members of the formed group.

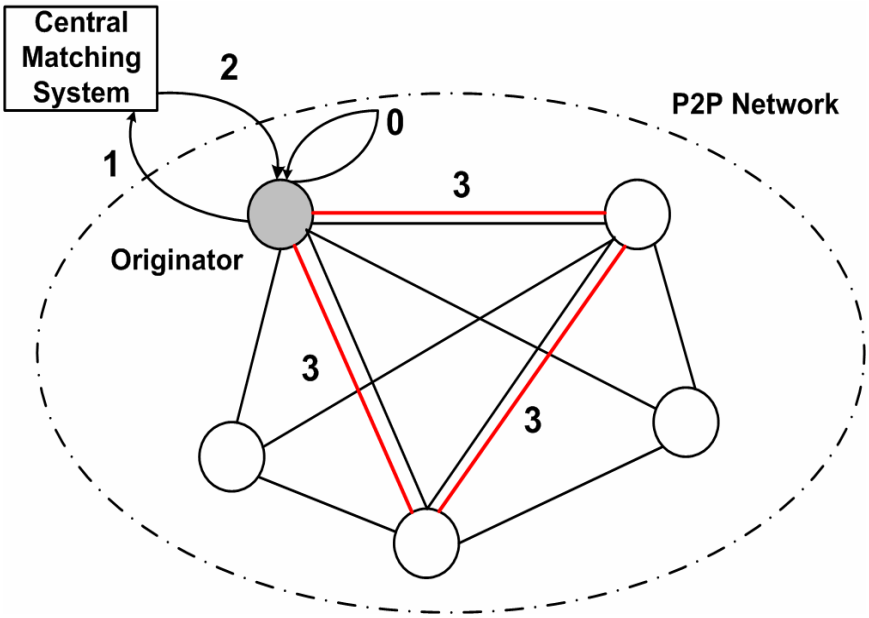

Figure 3: Formation of a peer group in response to a user query.

2.4 The PeerAssist matching and query system

The matching and query system of PeerAssist has the role of querying the available information based on the user inputs. With the exception of actually communicating with a peer, every other action that the user wants to perform using the PeerAssist platform is interpreted as a query that either retrieves information (for example Search Item scenario), or triggers changes of the stored information (for example Join Group or Leave Group scenarios).

The Matching and Query System's functionalities are as follows:

1. Formulate queries based on the user inputs & query templates

The users provide their inputs using the user interface, and this inputs are further transmitted to the Semantic Layer Agent (see section 4). A sub-component of the SLA will formulate queries based on this input, translating the captured user intent into a machine interpretable formalism, namely SPAQL queries.

2. Execute query against internal (local) semantic data

Every node stores locally information about the peer, and it is more optimal from both time and networking points of view to execute queries locally, when this is possible. A local matching system will be installed on every node, for executing those queries referring to locally stored data.

3. Execute query on the central data repository

There are two scenarios in which queries need to be executed against the central management system. Firstly, if the user performs some updates on its own node, these updates need to be propagated in the local repository (for example updating a peer's list of friends is performed both local and on the global repository). Secondly, is the user simply searches for data not available locally (for example a group with a certain interest).

4. Filter data based on additional criteria

An optional feature it will allow the filtering of the query results based on some preferences expressed via the User Interface (for example filtering the results based on age a peers, location, gender).

3 The User Interface layer

The PeerAssist User Interface comprises a set of hardware and software components to allow the user to manipulate the system. These components must capture the user's physical moves and translate them into logical actions that enclose the user intent. As defined in the requirements, the UI subsystem must be accessible and multimodal. The work on WP3 proposed an approach to address these features based on intent capture techniques and a Personal Assistant. This section describes the technical design for such a system.

3.1 Hardware deployment

The base framework regarding hardware deployment is represented in the following figure. The system runs on the PAnode in the user's home, and it is managed through several UI devices. These devices can be connected directly to the node, or they may communicate with it remotely within a home network.

Besides remote communication, it is intended to support a broad variety of devices (PCs, TVs, tablets, etc.). To allow that, it is chosen to use Web technologies to implement the UI subsystem.

The PAnode runs an HTTP server which interfaces the remote devices with other system modules. It delivers the UI contents as HTML (or VoiceXML) pages, and receives events from the user devices. On the other side, these devices run a web browser that connects to the server and renders the UI pages. This content includes JavaScript code to provide rich functionality on the client-side and asynchronous communication with the server.

The "hosted" devices are simple peripherals attached to the PAnode with some non-IP data link, e.g., a monitor, keyboard, mouse, speakers, etc. In these scenarios, the computer running the browser is the PAnode itself, which also runs the server. This deploy allows to have the whole system in a single machine (e.g. a PC), assuming it has multimedia capabilities.

In contrast, the "networked" devices are independent computers that communicate through the network to the PAnode, e.g. a tablet, smartphone or another PC. These machines run the browser locally and transmit data through HTTP. The term "web browser" here means both graphical (HTML) and voice (VoiceXML) browsers, which are functionally equivalent.

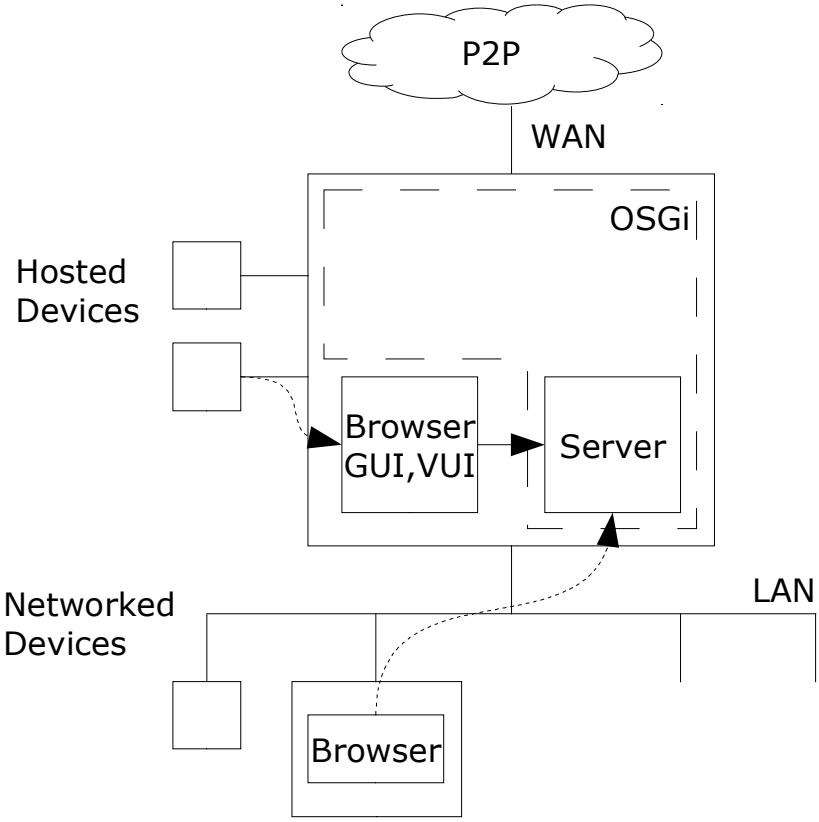

Figure 4: User interface deployment

3.2 Key solutions

Given the proposed functionality concerning multimodality and a Personal Assistant (see D3.1 and D3.2), the following key solutions have been considered to design a software module that supports it.

- **Explicit interaction models**
  - The UI workflows can be specified in a formal language (SCXML), which makes them independent from the modality used to interact with the user.
- **Standard and Assisted UI**
  - The standard Web interface is complemented (not replaced) with an assistant for an easier way of interacting, i.e. messages, questions and answers.
- **Graphic and Voice interactors**
  - The UI can be managed on screen devices or by voice, both for input and output. Even the two modes can be used at once.
- **Runtime adaptations**
  - The appearance and the contents of the UI can be dynamically adapted to the user based on their profile, context, etc.

The underlying concept is the ability to provide the user several alternative ways to perform the same task. Therefore it is needed a single component to drive the interaction processes, which can be accessed through many different UI modules. This design is inspired by the architecture proposed by the W3C Multimodal Interaction (MMI)

Working Group (http://www.w3.org/TR/mmi-arch/). The PeerAssist system will not implement it exactly; rather, its main ideas will be adapted to fit the requirements.

The multimodal architecture is conceived to support multiple modalities. In PeerAssist two modes are established, Graphic and Voice UI, which can be rendered on many devices. Besides that, the "assisted" UI is considered another modality which uses dialogs and questions instead of buttons, menus, etc. Thus, the system will offer Standard and Assisted UIs, supported by the same multimodal framework. This is orthogonal to the Graphic/Voice modes, i.e. the assistant messages can be displayed or spoken.

3.3 Software components

The following figure gives a high-level overview of the User Interface components:

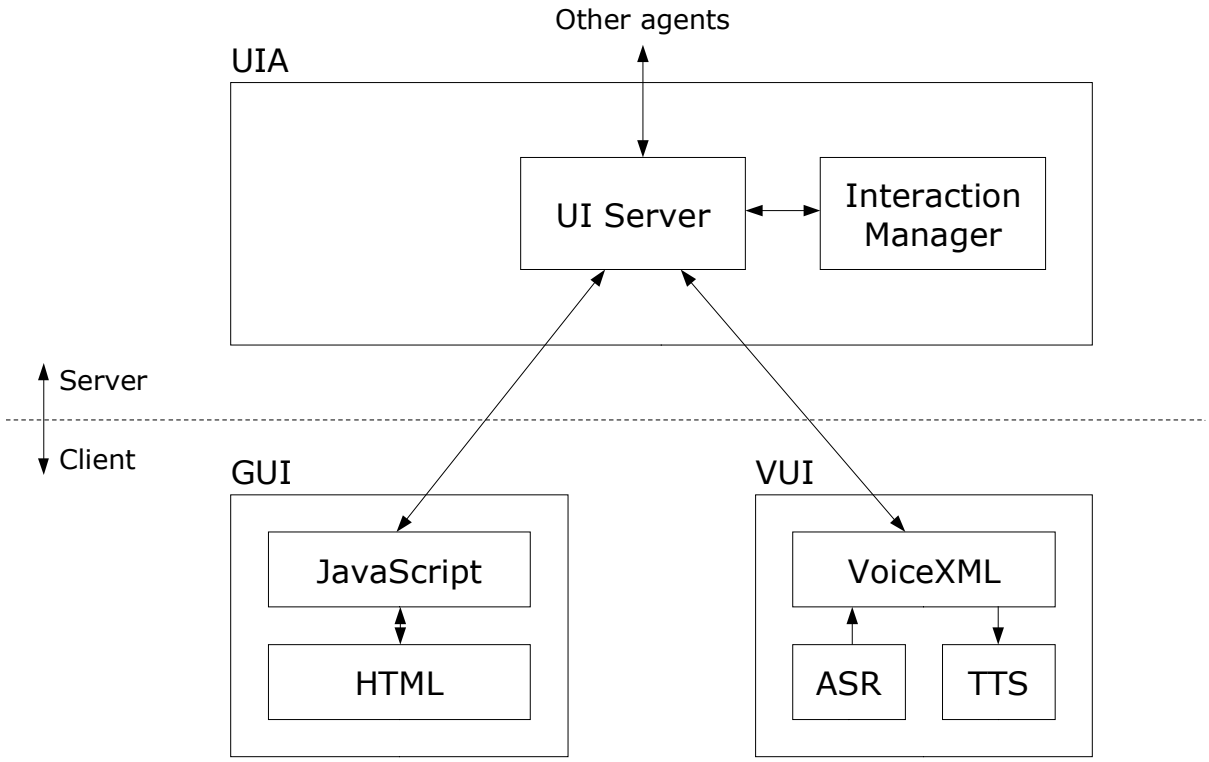

Figure 5: User Interface architecture

3.3.1 Interaction Manager

The "workflows" are sequences of steps the user must follow to perform a use case (e.g., make a group, search services). Each step involves some user interaction (displaying or entering data), as well as the execution of some work in other layers (e.g., send data to the network).

These workflows, or interaction processes, are expressed as **state machines** in the SCXML language, where each state represents a step. There can be nested sub-states, and even parallel sub-processes. The multimodal interaction is achieved by having several parallel paths rendered in different modality components (GUI, VUI). So, for example, a step of "enter search criteria" may have a sub-step for the GUI (show a HTML form) along with another for the VUI (ask questions). Then the user can interact with any modality, and the progress is **synchronized** between them.

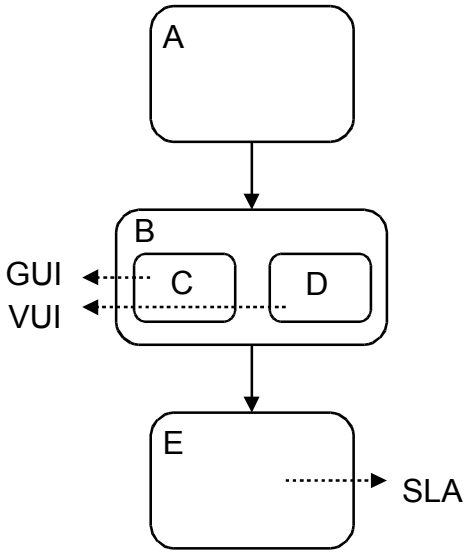

Figure 6: State machine and messages to other components

The Interaction Manager contains an SCXML interpreter which controls the progress of the workflow. For each step it sends content to UI components through the UI Server, and listens for user events. On the other side it interacts with other modules, also through the UI Server, e.g. invoking SLA or CA features with the user input, or receiving data from them.

3.3.2 UI Server

The UI Server works as a mediator between the Modality Components, the Interaction Manager and the other modules on the PAnode. Upon request, it generates the actual UI content, adapted for each modality. That is, HTML + JavaScript for the GUI, and VoiceXML documents for the VUI. This content is then delivered to each client, be it local or remote, through an HTTP interface.

As for data input, the UI Server receives user events and data from the clients by Ajax techniques. Then it passes them to the Interaction Manager or other agents.

Besides that, the UIA needs to send server-initiated events to the clients in order to notify them of the advances in the workflow and keep them synchronized. Since the HTTP protocol doesn't support that. the UI Server will use Comet techniques for that purpose.

3.3.3 GUI Client

The GUI Client runs on a Web browser to enable user interaction on the graphical modality. Like common web applications, it renders HTML content on the screen and executes JavaScript code to perform UI-related actions, such as updating the views and capturing user events.

It also contains logic to send these events and attached data to the server, as well as receiving push notifications from it and handle them properly.

This graphical client is responsible for displaying both the Standard and Assisted UI variations, i.e. render common widgets (buttons, links) along with text messages, questions and selectable answers, as shown in the Personal Assistant sketch (Figure 2 in D3.2).

3.3.4 VUI Client

The VUI Client runs on a Voice Browser for the speech-based user interaction. It contains a VoiceXML interpreter whose main job is to execute the dialogs coming from the server as VoiceXML documents. In general, these dialogs consist of saying messages, asking questions and waiting for answers.

The messages are spoken through a Text-to-Speech (TTS) module, which can synthesize arbitrary text. The user utterance is captured by an Automatic Speech Recognition (ASR) component, which translates it into a set of acceptable words. By combining these modules with the VoiceXML processor, it is possible to iteratively request the user a set of data, just like s/he would fill a web form in the graphical UI. This user input is then sent to the server.

Clearly, the voice modality must be based on spoken dialogs, as opposed to standard graphical widgets. Therefore it will only use the Assisted version of the UI. Ideally, the same messages played by the PA would be either displayed on-screen or spoken by voice, or even both at the same time.

3.4 Component interfaces

The UIA needs to invoke functionality on other agents in the PAnode through the PA. This communication involves sending and receiving application data. The interfaces are defined by the data that the agents need to exchange. All defined interfaces among PA node components are described in Appendix B at the end of this document.

4 The Semantic Layer

4.1 Semantic representation of PeerAssist concepts

For matching user queries with potential results, i.e. peers, groups, services, or content items, we make use of semantic representation of these entities and of technologies such as RDF and SPARQL for representation and querying (see deliverable D3.4 for details on query processing). In order to enable a functionality splitting between a single PAnode and the Central Matching System we chose to split the semantic representation into several different parts namely the PA (within the intelligence component) SLA (a query mechanism for accessing local and global semantic information) and CMS as well as two adjacent ontologies, namely the general and context ontologies.

PeerAssist General Ontology

This is the main ontology at the core of the PeerAssist semantic layer and defines the shared general concepts regarding User, Group, Content, Preferences and Services and relations among them. Deliverable D4.3 gives a detailed overview of the main classes of the general ontology and how they are related to each other. The PeerAssist context ontology makes use of the concepts defined in this general ontology. As discussed in deliverable D3.3, we reuse concepts from existing ontologies, namely FOAF (denoted using the foaf: prefix), SIOC (sioc: prefix) and GoodRelations (gr: prefix), and extend these ontologies with concepts specific to PeerAssist. For example, we modeled the User concept as a subclass of both foaf:Agent and gr:BusinessEntity. This allows us to reuse properties from FOAF for describing user profiles, such as age, interests, location, as well as properties from GoodRelations like

the gr:offers relation to represent that a peer offers a service. The SIOC ontology is mainly used for representing content items (sioc:Item concept) and containers for these items (sioc:Container concept) such as discussion forums or personal blogs. The general ontology resides partly on the SLA as well as fully on the CMS.

PeerAssist Context Ontology

This ontology captures the context of a user within the PeerAssist system It comprises things like habits, areas, day times and notifications. It is needed in order to derive new knowledge based upon temporal and geographical conditions. Central to the context ontology is the concept PeerAssist User which it inherits from the general ontology. The context ontology resides on the PA, the PA intelligence component to be exact. User context should be only stored locally and is only relevant for a single user in order to perform machine learning supported system and interface improvements based upon the specific user's behavior. Thus we decided on not sharing contextual information through the CMS to be queried by other peers. Actual user searches are stored within the PA as part of the context information.

These two ontologies and their common interaction are described in detail in deliverable D4.3. The next section will outline the overall architecture of the Semantic Layer.

All ontologies and their common interaction are described in detail in deliverable D4.3.

4.2 Semantic Layer Architecture

The Semantic layer consists of a number of sub-components responsible with building queries, executing them, providing the response to the user and maintaining the ontologies. The queries are constructed based on the input received from the user interface (namely transforming the user actions into a query) or the Personal Assistant, a task performed by a Semantic Layer subcomponent called *Query Builder* (QB). Depending on the goal of the query, it will be further executed either by the *Local Matching System* (LMS), a software component deployed locally or by the *Central Query System* (CQS). The CQS is part of the *Central Management System* (CMS), the main comportment of the Semantic Layer.

The overall Semantic Layer architecture is as follows:

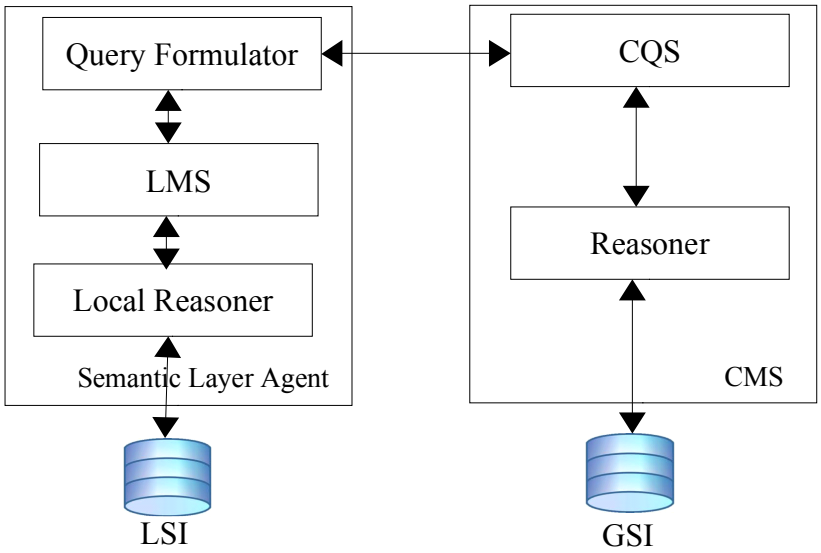

Figure 7: The Semantic Layer Architecture: (left) The SLA component architecture

4.3 Semantic Layer Components

The Semantic layer includes the **Semantic Layer Agent (SLA)** that is part of the PAnode software architecture and runs at the end-user's PC and the **Central Matching System (CMS)** that is outside the P2P network and holds the General PeerAssist ontology and query matching engine.

The Semantic Layer Agent consists of the following components:

Query Formator – the query building component receives the user or Personal Assistant inputs and transforms them into a query. The same sub-components will return the results to the invoking party. If the queries can be solved locally, they are further sent to the Local Matching System. If the query is addressing data not maintained locally, it is sent for processing to the Central Matching System.

Local Matching System (LMS) – a matcher entity that has the task of executing local queries, against the Local Semantic Information (LSI) repository. These queries are referring to a peer's own data. The result of the query is sent back to the query building component which has to forward it to the external component, and if needed to trigger updates in the General ontology that resides in the Central Matching System.

Local Reasoner- It will provide support for executing local queries. Being deployed at the peer, it should not be a fully-fledged reasoner (due to resource and optimization constraints) but rather a light version of a reasoner adapted for the information stored locally.

Local Semantic Information (LSI) – contains those part of the global PeerAssist ontology relevant for the peer (e.g. his own information, information about his friends, doctors, and third party services most often used) as well as the context and suggestion ontologies.

The Central Matching System (CMS) is the main query system of the PeerAssist architecture, and consists of the following subcomponents:

Central Query System (CQS) – it has the task of executing queries addressing against the entire PeerAssist information.

Reasoner – will provide support for execution the queries. As opposed to the local reasoning support, the one deployed at the CMS level should have full reasoning and inference capabilities.

Global Semantic Information (GSI)– an RDF repository that contains the PeerAssist general ontology

SESAME1 repository and reasoning support will be used for implementing the CMS. The rational for this choice, as well as a more detailed description of the semantic layer components will be presented in deliverable D4.3

5 The Communication Layer

This section describes the functions provided by the Communication Agent (CA) of the PAnode. The overall architecture of the CA is presented in Figure 8. The CA handles all peer-to-peer communications as well as security aspects as described bellow. The agent also interacts with the rest of the agents through the Personal Assistant (PA). The UIA captures user intent, forms user queries and presents results. The SLA maintains user profile and context information, as well as interacts with the CMS for semantic matching. Finally, the PA mediates whenever needed to facilitate interaction with the end-user.

The CA provides to the PeerAssist platform and the interacting agents (i.e., UIA, SLA and PA) the core P2P JXTA services, and the required network services and supports the provision of PeerAssist applications (e.g., raise an alarm).

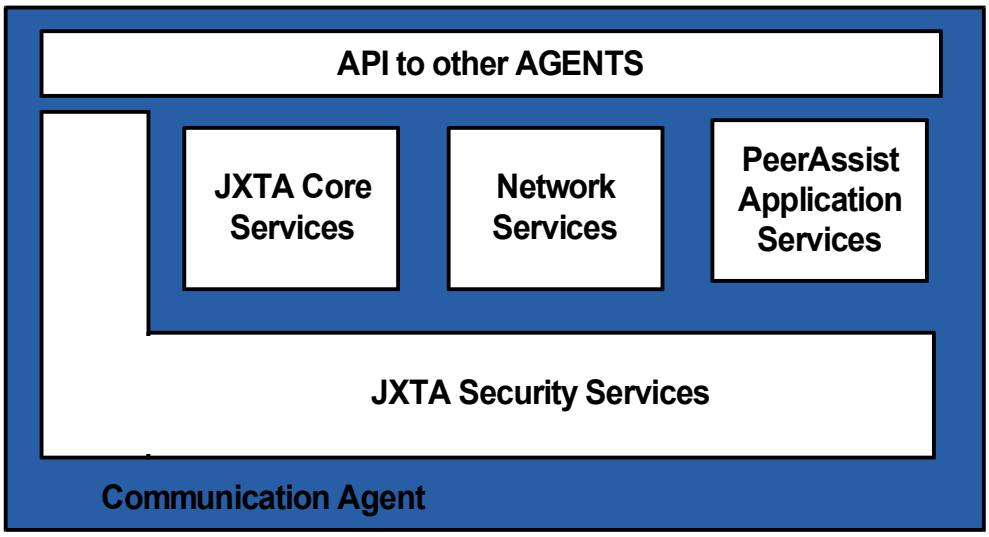

Figure 8: The overall Communication Agent architecture

5.1 Security functionality

1. **Subscribe a peer to the system:** This function will assign a unique identifier (ID) to the peer. It will generate the peer's certificate (public & private key). The private key will be safely stored in the peer's device, while the public key will be uploaded in the system's list of peers. It will enable the peer to select a nick name for using the platform and set its credential (i.e., password).
2. **Unsubscribe a peer from the system:** The peer will be authenticated to the system asking to be removed from the system and system's list of peers. The peer's certificate is uploaded to the certificate revocation list and the peer's profile is removed.
3. **Modify peer's subscription:** The peer may change its password or generate a new certificate. The old certificate will be uploaded to the certificate revocation list.
4. **Authenticate a peer to the system:** This function enables a peer to be authenticated to the system using either its user name/password or its private key.
5. **Mutual peer authentication:** A couple of authenticated peers are mutually authenticated each other using digital certificates.
6. **Provide a secure channel:** After mutual authentication, the system provides a secure communication channel between two peers. The granted security services may include: confidentiality, integrity, authenticity and acknowledgment of receipt.

5.2 Grouping functionality

1. **Create a peer group:** This function enables a peer to create a group. The created group is assigned a unique group ID and a short text describing its goal. The initiating peer is included in the created group. The group can be public or private.
2. **Delete a peer group:** This function erases a group. The group to be deleted is identified by the unique group ID. Only the initiating peer can erase the group.
3. **Join into a peer group:** This function enables a peer to join into a group. The group is identified the group ID and the peer by by the peer ID.
4. **Resign from a peer group:** This function enables a peer to resign from a group.

5.3 Service functionality

1. **Create a communication channel between peers:** This function enables the establishment of a communication channel between two peers. The created channel is uniquely identified by an ID. A short text describes the channels goal.
2. **Delete a communication channel between peers:** This function deletes a communication channel between two peers.
3. **Push data to a communication channel:** This function pushes data (message or a sequence of messages) originating from a related application and a specific peer to the created communication channel. The created communication channel is identified by the unique channel ID
4. **Receive data from a communication channel:** This function receives data from the communication channel and forwards them to the related application of the specific peer.

6 The Personal Assistant Architecture

The Personal Assistant (PA) is a critical component of the PeerAssist architecture, and is responsible for the coordination of the rest of the platform components. The PA provides a single point of integration, allowing the other components to operate more or less independently from each other. By implementing the business logic of the PeerAssist system, the PA helps realizing the various use cases by properly orchestrating the operations offered by the other components, thereby delivering the required advanced functionality. At the same time, the PA logs all message exchanges and interactions among the PeerAssist components, for the purposes of history tracking and troubleshooting.

The PA is responsible for asynchronously interacting with the end-user, mainly through a robust notification mechanism, which is powered by the intelligence acquired through close monitoring of the end-user's environment by means of sensor devices. The PA hides all technical complexity from both the end-user and the remaining components of the PeerAssist platform, and thus contributes significantly to the maintainability of the system.

In the remainder of this section, we present the design and capabilities of the PA component in more details.

6.1 The Personal Assistant Module

The PA is a central module that communicates with the agents: CA, SLA, and the UIA on the PeerAssist node. The PA provides:

(i) a dispatching functionality to other agents, i.e., CA, SLA and UIA, and

(ii) an extensible intelligent functionality to materialize context-awareness.

The Intelligent functionality of PA refers to capturing sensor-based events in order to notify/warn/alarm care-givers/other authorized PeerAssist users. Specifically, PA relays on an inference engine, which is based on ontological context modeling and representation. This inference engine can be easily extended to more complex methods/techniques, e.g., machine learning algorithms for profile adaptation. The intelligent functionality is supported through the *Reporting* and *Triggering* interfaces (discussed later). The Dispatching functionality of PA relays (dispatches) messages (calls) from UIA to CA and SLA and from CA to SLA and UIA. The PA initiates messages (calls) to UIA and CA in order to support the intelligence functionality. Moreover the PA captures messages (calls) from CA for certain functionality (e.g. invitation of the user). The dispatching functionality is supported through the *Dispatching* interface (discussed later). Figure 9 shows the position of the PA with respect to the other agents. One can observe the interfaces with the other components. Each agent implements its interface (UIAInterface, SLAInterface, CAInterface) through which it communicates with the PA.

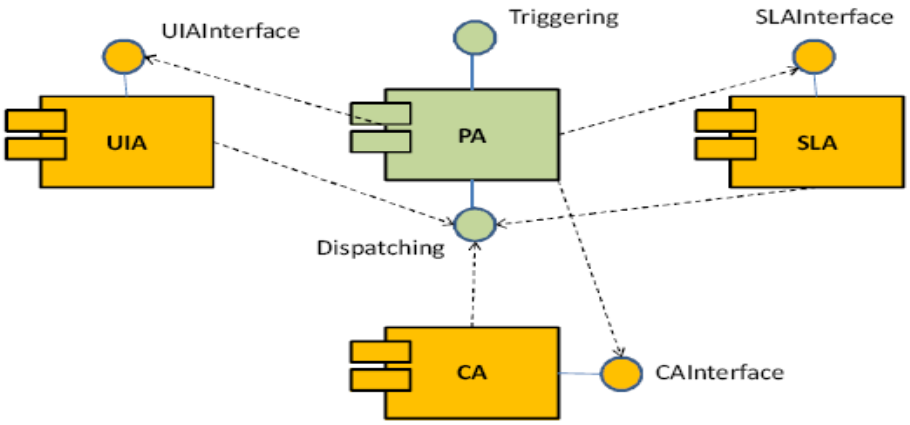

Figure 9: The PA's interaction with the CA, UIA, and SLA components via exposed interfaces.

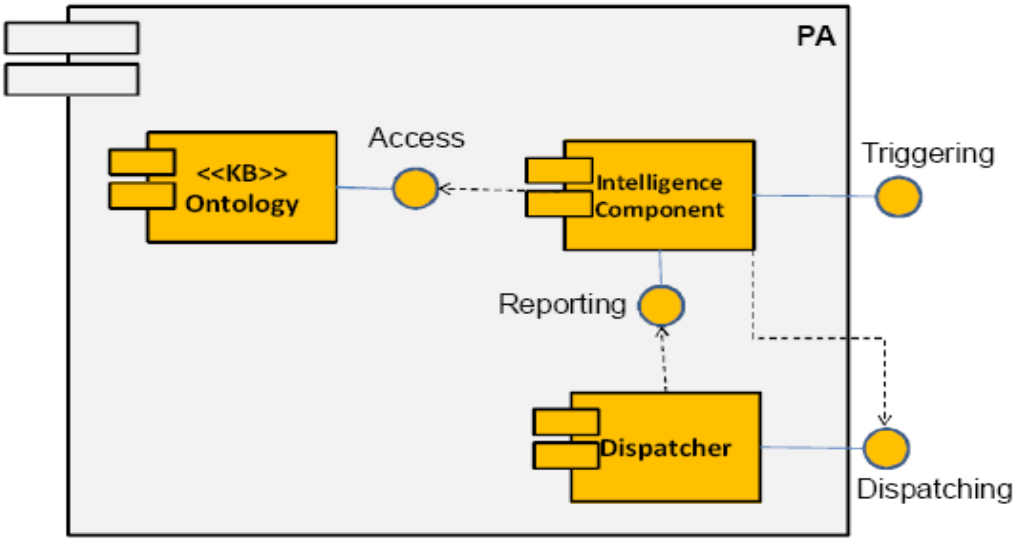

Figure 10: The internal organization of the Personal Assistant component.

The internal components architecture of the PA is shown in Figure 10. The PA consists of the three basic components: (i) the Ontology component, (ii) the Intelligence component and (iii) the Dispatcher component. The ontology and intelligence components comprise the intelligence part of the PA. In the remainder we elaborate on each component in order to explain the intelligence and dispatching functionality of the PA.

6.2 The intelligence functionality of the PA

The Intelligence part of PA consists of the Ontology component, which maintains a Knowledge Base (KB). It contains a set of inference rules for activating certain actions. The inference rules are based SWRL and the contextual ontology is represented through OWL. The concepts of the ontology are instantiated through the *Access* interface. This interface is responsible for managing the inference rules and contextual parameters. The ontology of this component refers to the Contextual Ontology which represents contextual information (i.e. time, space) and a set of inference rules. Figure 3 shows some part of the contextual ontology.

The intelligence functionality of the PA is triggered by the context sources from the Home Automation Controller (HAC). Through the HAC, the PA is being triggered from various sensors, e.g., the panic button sensor and the vision sensor. The HAC sends the signals from the sensors to the PA by using the Triggering interface. Then, the PA based on the current context of the user and the corresponding signals from the sensors acts as indicated by certain actuation rules. The following figure illustrated the HAC component and the PA component communicating through the Triggering interface.

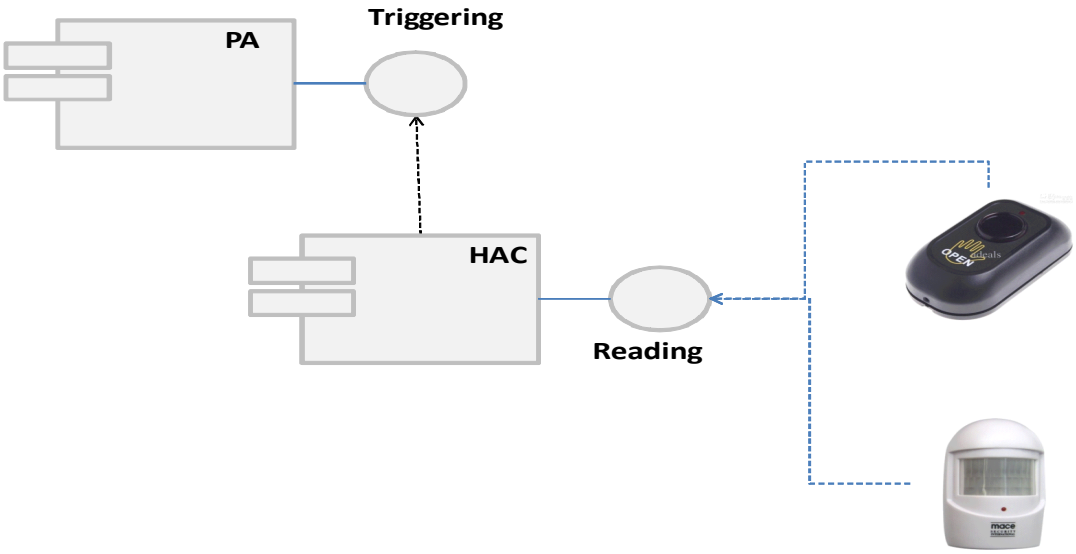

Figure 11: The communication between PA and HAC for handling sensors' readings

The signals from the sensors are translated into ontological instanced in order to fire the corresponding actuation rules. Specifically, based on the contextual ontology of the Ontology component of the PA, we can define certain context inference/actuation rules. Such rules are fired once the antecedent parts hold true. The antecedent part of a rule refers to certain contextual parameters. The PA intelligent component is responsible for the following rule (in linguistic view):

R1: "The PA warns the caregiver of a User once she is not detected in the living room in the morning, as she used to be".

This rule refers to space (lining room), time (morning) and habits of a User. A motion detection sensor provides information related to the exact location (e.g., room of the house) of the User. Once the user used to passing her morning time at the living room and, for a certain period of time, she is not detected on the living room (at the morning) then the rule is fired. Upon firing a rule the PA sends notification messages (e.g., warnings, alarms) to the user's caregivers group, or other authorized persons (e.g., relatives etc.). Such rule can be expressed through the following SWRL rule:

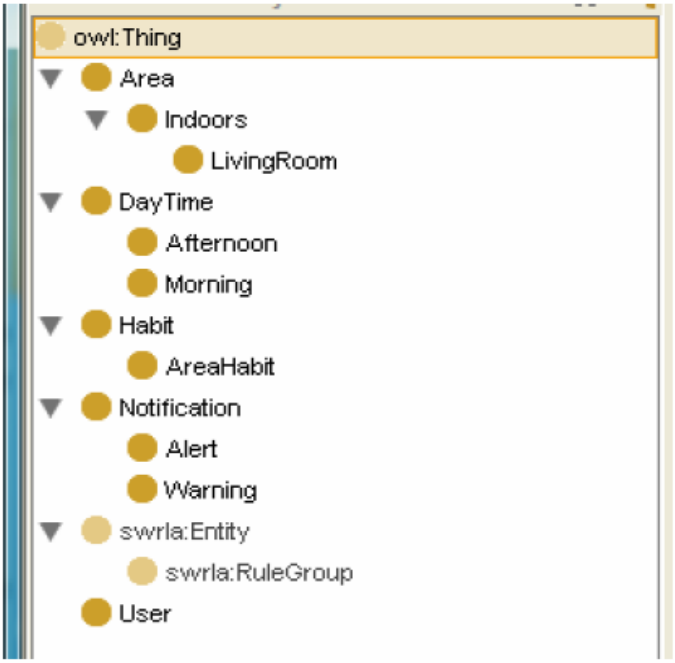

Figure 12: A snapshot of the contextual ontology

User(?u) ∧ Area(?l) ∧ isLocatedIn(?u, ?l) ∧ hasHabit(?u, MORNING_LIVINGROOM_HABIT)

∧ hasCurrentDayTime(?u,MORNING)

∧ differentFrom(?l, LIVINGROOM) → isBeingNotified(?u, WARNCAREGIVER)

Such rule is triggered from signal obtained from the vision sensor. Such SWRL rule is constructed by concepts and relations of the context ontology. The Intelligence part of PA consists also the Intelligence component, which is triggered by contextual information of the user's environment. The Intelligence component infers the user situations based on the inference rules from the Ontology component. In this case, it uses the *Access* interface. The intelligence component provides the *Triggering* interface for capturing external sensor events. Moreover, the following SWRL rules are also supported by the PA, thus, materializing the so called context-awareness:

R2: "The PA notifies the caregiver of a User once she moves in a specified area of the home, every T time instances".

R3: *"The PA warns immediately the caregiver and doctor of a User once she pushes the panic button".*

The R2 rule specifies a periodical message generation with frequency 1/T, which is per-determined by the caregiver, e.g., T = 1h. Such rule is supported by the Ontology of the intelligent component of the PA and, in SWRL format, has the following syntax:

User(?u) ∧ Area(?l) ∧ isLocatedIn(?u, ?l) ∧ timeOfDifference (?u, T)→ isBeingNotified(?u, NOTIFYCAREGIVER)

The R3 rule is a very important rule since once the user pushes the panic button sensor then the caregiver and any assigned doctor are immediately warned implying an urgent event. This rule is supported by the Ontology of the intelligent component and, in SWRL format, has the following syntax:

User(?u) ∧ isPushedPanicButton(?u) → isBeingNotified(?u, URGENTCAREGIVER) ∧ isBeingNotified(?u, URGENTDOCTOR)

Moreover, the PA provides the *Reporting* interface for recording all message sequences among the agents for acquiring a holistic view of the user's behavior. It uses the *Dispatching* interface for communicating with other agents (e.g. sending a warning message to UIA).

6.3 Dispatching functionality of the PA

The Dispatching part of PA consists of the Dispatcher component. This component relays messages among other agents and reports the sequence of messages to the *Reporting* interface for recording the behavior of the user. It is worth noting that the reporting interface can be used for extending the Intelligence functionality of the PA, e.g. through Machine learning algorithms and/or learning novelty inference rules that reflect the user's pattern and communicative preference. The external agents, i.e. CA, SLA and UIA, use only the *Dispatching* interface to realize inter-agent communication. Some basic methods for the Dispatching interface are:

- notify() -- from PA to UIA
- inviteUsers() –from UIA to PA and then from PA to CA
- joinGroup() –from UIA to PA, from PA to CA and SLA
- createGroup() – from UIA to PA, from PA to CA and SLA
- deleteGroup() -- from UIA to PA, from PA to CA and SLA
- leaveGroup() -- from UIA to PA, from PA to CA and SLA
- notifyGroup() -- from UIA to PA, from PA to CA and SLA
- …

The complete list of the interfaces can be found at in Appendix B.

The Intelligence functionality of the PA can be extended (i) based on the *Reporting* interface through which one can monitor/record/observe the message sequences among agents, and (ii) based on the *Triggering* interface through which one can capture contextual values from the user's environment. The PA Intelligence component can extend its functionality by Developing Machine Learning algorithms (e.g. learning user preferences) and by asserting easily new inference rules through the Ontology component.

7 The PeerAssist platform architecture

7.1 Platform components & design constraints

The use cases presented in deliverable D2.3 identify the need for the following services:

- *User interfacing:*

- Graphical user interface, providing a visual mechanism for user interfacing: The user is able to control the system using a visual tool, using buttons or menus to select actions or monitor another user with video streaming
- Voice interface, providing a voice-enabled mechanism for user interfacing: The user is able to control the system using his voice and hear the result of his actions or indications about monitored activities.

- *External communication with other PeerAssist nodes*
  - Peer to peer framework, for communicating with other PeerAssist nodes using peer to peer technologies.
  - External network interface (Wi-i or Ethernet) with broadband connectivity
- *Control and monitoring of in-home devices:*
  - Home automation framework, for control and monitoring of devices and equipment inside the home. The framework must provide support for the selected home devices.
  - Home automation network interface. The interface must provide the connectivity with domotic devices.
- *External Central Matching System:*
  - Acts as a service broker for additional services support, content search and additional queries processing when needed
  - Interface for communication to the Service Broker providing a central entity for services support, additional content search and additional queries processing when needed
- *Service management platform*
  - A platform for management of home platform software and services

Analyzing the services mentioned above, a number of system components and interactions are identified, providing a first view of the PeerAssist platform architecture, as presented in the following figure.

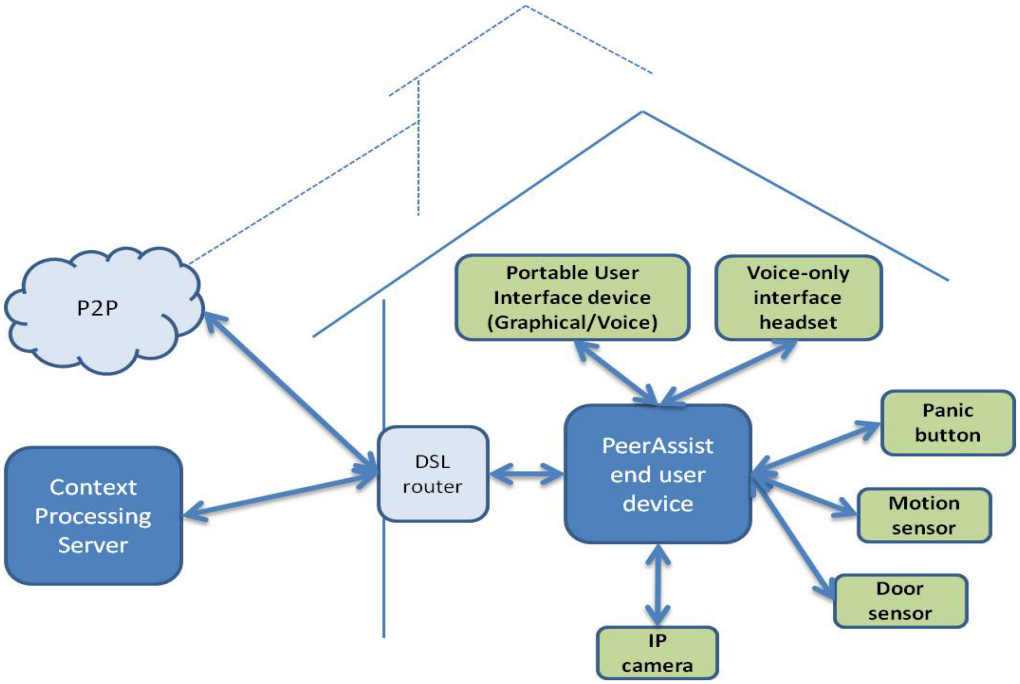

Figure 13: The PeerAssist Home Platform

The central point of the home network is the **PeerAssist end user device**, running the PAnode software, providing home services, interacting with the user and managing the home infrastructure. This local server is a simple Personal Computer (desktop or laptop) or an embedded device. In the case of the PC, the server is the same with the end user device, as the web application has minimum hardware requirements to run on the server.

The **Portable User Interface Device** is the system for interfacing most of the platform functionalities to the user through communication with the PeerAssist end user device running the PAnode software entity. It can be either a tablet, or a laptop, and it will connect to the PeerAssist end user device (in case they are two separate devices) through a graphical user interface or a voice user interface, using a headset or a microphone and a set of speakers. The Portable User Interface Device may also be part of the PeerAssist end user device (integration into one device).

Added value services may be enabled by the use of several home peripherals: panic button, motion/door sensor, camera, microphone. The **Panic button** is a simple button that the user can push when s/he wants to trigger an alarm. It is connected to the local server and when it's pushed, it triggers an alarm to notify the user's caregivers. The **Camera** is connected to the local server and it is used in the use cases of monitoring activities or making video calls between two users. The camera is connected to the local server using a wireless interface if it has one. The **Motion and Door sensors** are used to provide context monitoring, enabling the monitoring of specific user activities and home status.

The devices such as the camera, speakers, microphone and panic button, are either connected to the local server (end user device) directly through cable, or through Bluetooth if it is supported, or even through the wireless network (also if it is supported). There is no real difference, except the fact that if they are connected on the wireless network, they can be used by different software components of the system and not only by the application that runs on the end user device.

The graph of PAnodes is formed using a P2P overlay network (JXTA in our case) as shown in the figure below:

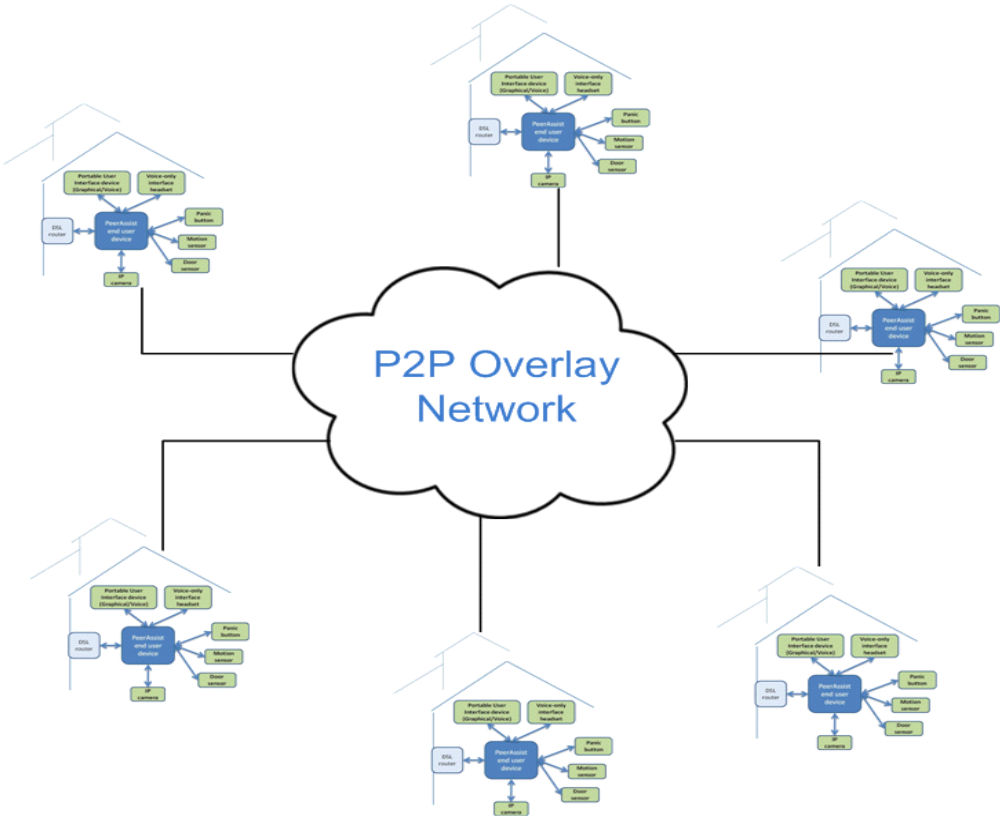

Figure 14: The PeerAssist network of nodes

7.2 Home platform design and architecture

A detailed analysis of the use cases, requirements in conjunction with the components identified above and the design constraints imposed, has provided a first level of system architecture for the end user device, presented below.

The main components of the End user device are the following:

- **Graphical User Interface**

  The Graphical User Interface is the main way the user interacts with the system. The GUI is a web interface, as the application is a web application with a Java backend.

- **Voice User Interface**

  The user can perform the same actions he can perform with the web interface, using only voice commands and receiving voice as feedback from the system afte every action. The two interfaces are always synced, so the user can use them both in parallel.

- **HTTP Server**

  The HTTP server hosts the web application. It runs on the OSGi framework and its only purpose is to serve the front-end to the user and communicate with the back-end Java application.

- **Interaction Manager**

The Interaction Manager is responsible for the synchronization of the graphical user interface with the voice user interface. Every time the user performs an action with one of them, the interaction manager informs the other about the action and changes it's state accordingly.

- **Personal Assistant (PA)**

  The Personal Assistant is a software module that helps the user perform the actions he wants, either by providing him with information about each action, either by learning the most frequent user's actions, and making it easy for the user to perform them again. The personal assistant is described in detail in another paragraph.

- **Communication Agent (CA)**

  The Communication Agent is responsible for the communication between the users of the system. It receives the information about the other users from the broker interface, and exchanges messages between users and user groups.

- **Semantic Layer Agent (SLA)**

  The SLA is responsible for the communication with the Central Matching system (CMS). It synchronizes the information stored locally, with the information stored on the CMS It also retrieves the information about the other peers of the system from the CMS.

- **Home Automation Controller (HAC)**

  The Home Automation Controller is responsible for the control of and communication with local devices and sensors (e.g., panic buttons, motion sensors, etc.) used in certain PeerAssist use cases.

7.3 System architecture and integration

7.3.1 End user device

Evaluation of specific technologies from WP3 has led to the selection of specific solutions and design choices, an overview of which is presented in the following figure.

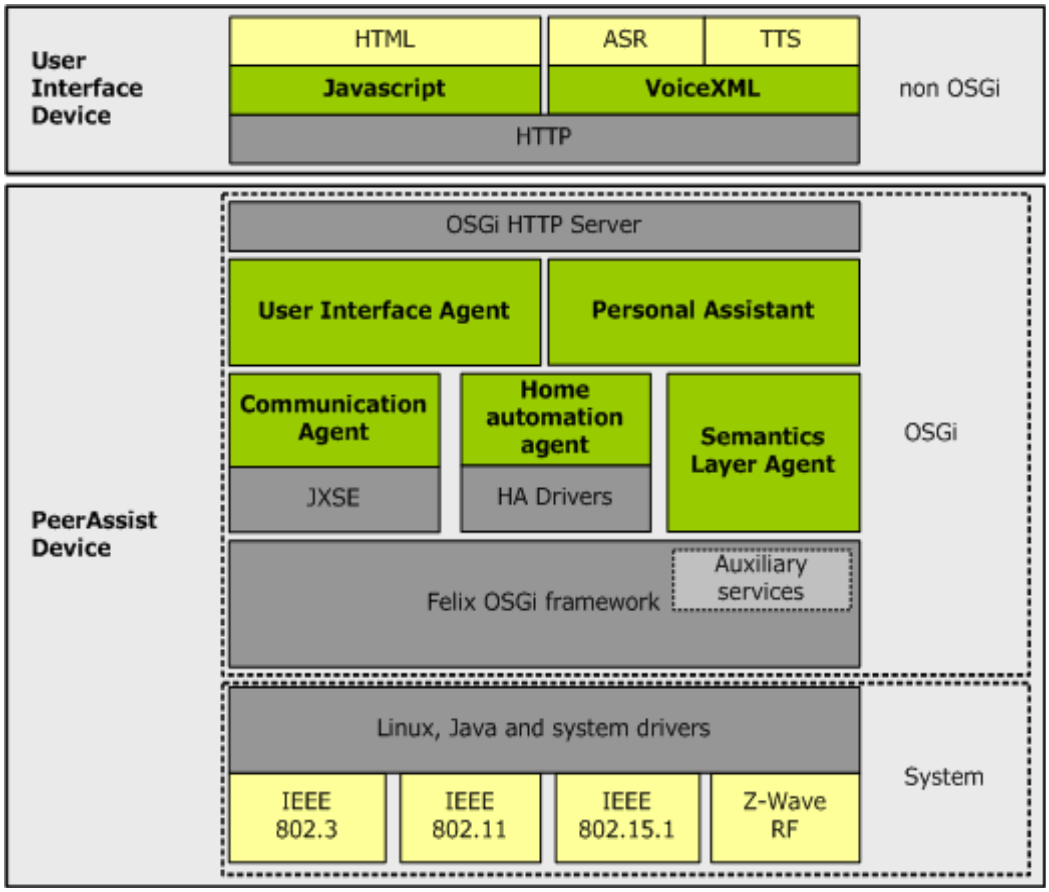

Figure 15: The PeerAssist software components mapped to the PeerAssist platform.

The end user device runs a web server with an integrated application server, *Jetty*, a common and lightweight web server that has minimum hardware requirements. The application server hosts the main application that is responsible for the interaction of the user with the platform. The end user device also runs a VoiceXML browser, that hosts the voice xml scenarios for the voice user interface. The end user device finally hosts the ASR and TTS modules, that are used by the voice user interface, for the voice recognition of the user voice commands and the response to the user. The VoiceXML browser is hosted on the same machine as the application server, but they communicate through http calls so they could be hosted on different machines or even different platforms. The ASR and TTS modules, are not part of the VoiceXML browser, but they are third party components serving the need for different languages rather than the default English language that is integrated in the browser. The scenarios supported by the VoiceXML browser, include the same actions that the user can perform by using the graphical user interface, so that the user can either learn one from the two different options and use the other the same way, or during an action he can switch back and forth from one option (graphical or voice) to the other.

The P2P network is implemented by JXSE, the Java implementation of the JXTA framework. The JXSE network sits on top of the Apache Felix, a Java implementation of the OSGi framework, which is responsible for the service updates of the system, and the advertisement and broadcasting of the services available. Every end user device in the system, is an edge peer of the JXTA network, and runs a local installation of the Apache Felix. The edge peers are behind firewalls, and the communication between them is possible by using rendezvous peers, special purpose peers which are in charge of coordinating the peers in the JXTA network and provide the necessary scope to

message propagation. The traversal of the firewalls is possible by using relay peers, which allow the peers behind firewalls or NAT systems to take part in the JXTA network.

The portable user interface device, a tablet or a laptop, is the main client of the web server that runs on the end user device. In some cases, the end user device and the portable user interface device may be a single machine, but the architecture is exactly the same in both cases.

The portable user interface device is connected to a camera, a microphone and a set of speakers, or to a headset. The user can operate the device either with the mouse and keyboard or with voice commands, by using either the graphical or the voice user interface. The camera is used for monitored activities or for video chat and conference, and it is either directly connected to the portable user interface device or connected to the wireless home network.

The panic button is a simple button that is either connected directly to the end user device, or connected to the wireless home network. It is used in emergency situations, where the user wants to inform the group of people he has chosen to be informed in case of emergency (doctors, close relatives, neighbors). When pressed, the normal flow of control is interrupted and the users are informed that the user who pressed it is in danger. The button sends a signal to the application that runs on the end user device, and the users that are in the group are informed by visual and voice alerts. The use of the panic button is not very different than a normal message sent to a group of people, except that the message is transferred to the users in every possible way, and it doesn't stop being sent until the user responds or the user that initiated the alarm, stops it in some way (e.g. by pressing the button again).

The PeerAssist end user device is the central server of the home network, that gathers the input from all devices and does all the processing. Every device in the home network connects to the end user device, so that the last acts as the single point of access to the P2P network. The end user device has access to the internet and to the PeerAssist network, by connecting to the router inside the home network, through a wireless interface. The end user device can be a laptop, a netbook or in most cases a tablet PC, with minimum hardware requirements if there is another server that can host the applications, or more hardware capabilities if the end user device also hosts the applications and the voice browser.

The PeerAssist end user device acts as an edge peer to the P2P network. It runs an instance of the Apache Felix OSGi framework implementation, and it hosts all the applications that need to be accessed as OSGi bundles. It also runs a web server (as an OSGi bundle or as a standalone server), that hosts the web applications that must run locally in order to use the equipment (microphone, speakers) that connect locally to the end user device. The web server sits behind the firewall and it can be accessed only from within the home network, for security reasons and because there is no application that needs to be accessed from outside. The applications that run on the web server, can access the internet and the P2P network only through the OSGi interface and the JXTA network, by communicating with the rendezvous and relay peers, that have the ability to communicate through firewalls.

The end user device also has a built in camera, or a camera connected through a wireless interface. The camera is used in the use cases when the user must either talk to other users, share real time video, or perform a monitored activity that will trigger the alarm in the case of something visual, or if the monitoring is being done only by someone watching the whole activity of the user through the camera.

7.3.2 Voice platform

The Voice User Interface helps the visually impaired users and the users with limited abilities in using the normal interface. It consists of a microphone connected to the end user device, a set of speakers, also connected to the end user device, the ASR and TTS engines, the VoiceXML browser, a web server and a sip phone.

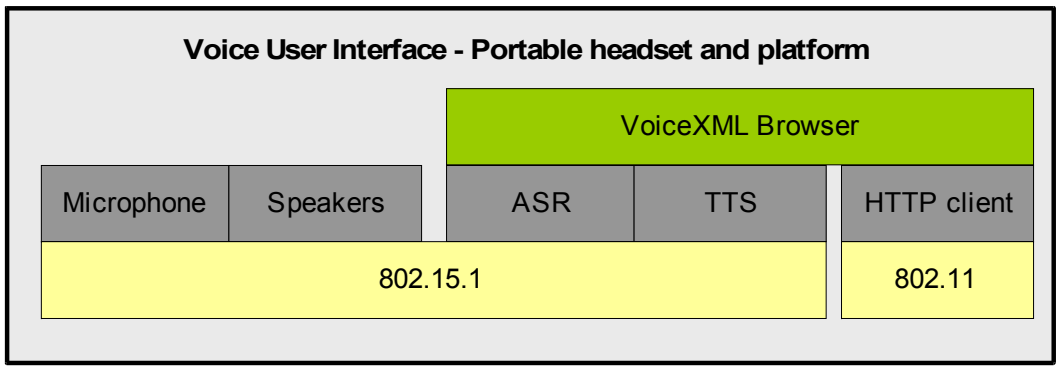

Figure 16: The voice platform and its components

- **Microphone**

  A simple microphone, connected to the end user device (the tablet or laptop). The microphone can connect either by cable or by Bluetooth and it will be always on, regardless of the availability of the graphical user interface. The microphone can be replaced with a headset, so that there is no need for the speakers in the case of the direct communication of the user with the platform.

- **Sip phone**

  The sip phone is a command line application that runs on the end user device, without the user's intervention. When the application starts (when the user turns on the end user device), the sip phone initiates a sip call to the VoiceXML browser. The sip call is the way the voice is transferred from the user to the server and back. The sip phone is responsible for maintaining the call, and in cases when the call is dropped, the sip phone will automatically initiate a new call to the VoiceXML browser, without the user being aware of that.

- **Speakers**

  A simple set of speakers connected to the end user device, either by cable or by bluetooth. The users listens to the speakers either for voice responses to the voice commands he sends through the voice user interface or for help and suggestions, sent to the user by the Personal Assistant. The speakers are always on when the end user device they are connected to is on, and they can be used in many use cases, even for monitored activities, if the VoiceXML browser is set to respond to either specific voice commands, or absence of voice for a specific amount of time.

- **ASR**

  The Automatic Speech Recognition (ASR) engine, is responsible for receiving voice from the user and translating the voice to text, so that the application running in the background, can transform the user's

voice commands to events, as it receives as input text commands that can be interpreted in an easy way. The ASR engine can understand voice commands and sentences in either Greek or Spanish language without configuration on the server, as it has built in capabilities of understanding the language spoken by the user and choosing accordingly the right grammar for the translation of the input. The ASR engine runs on the same machine as the VoiceXML browser and the TTS engine. Thanks to the most advanced neural network technologies, along with excellent robustness to background noise, the ASR engine used guarantees high performance even on large-scale vocabulary speech, whether using speech grammars or statistical language models. All this enables a dialogue with the user which is simple and natural. In a word, human.

- **MRCP Server**

  The MRCP (Media Resource Control Protocol) Server is an intermediate module that sits between the ASR engine and the VoiceXML browser. The MRCP server comes in two versions, that are mutually exclusive. The difference between them is the protocol they rely on for their communication. Version 1 server uses RTSP (Real Time Streaming Protocol) and version 2 uses SIP (Session Initiation Protocol). In the case of PeerAssist, the only option is version 1 of the server, as this is the only version that can connect to the VoiceXML browser the end user device hosts. The client (VoiceXML browser) requiring speech processing resources contacts the MRCP server over TCP and uses RTSP and SDP (Session Description Protocol) to set up a session. Once the session as been set up, two separate channels are opened: a media channel for audio data streams, which uses the Real-time Transport Protocol (RTP) and a control channel, running on TCP, in which messages in MRCP format are exchanged. SIP messages are text messages sent over the regular internet (i.e., TCP). As with most text-based internet message formats, SIP messages consist of a first line, a header and an optional body. The body of a SIP message is in SDP format.

- **TTS**

  The TTS engine is responsible for translating text to speech, that the user can hear from the speakers. The application that runs on the end user device, receives the input from the user's voice and responds with predefined or dynamic answers, according to the user's intent or the input from the server. The TTS engine is capable of translating text to either Greek or Spanish language, according to the language of the user. The synthetic voices that are used, offer a rich variety of advanced features: they are expressive and natural-sounding, they can read in several different styles, they are able to cry and to laugh, and they can correctly interpret text message abbreviations, acronyms and emoticons.

- **VoiceXML Browser**

  VoiceXML (VXML) is the W3C's standard xml format for specifying interactive voice dialogues between a human and a computer. It allows voice applications to be developed and deployed in an analogous way to HTML for visual applications. Just as HTML documents are interpreted by a visual web browser, VoiceXML documents are interpreted by a voice browser. The voice browser receives the user's input from the microphone, establishes a connection with the MRCP server in order to transform the user's voice to text, and finally responds to the user again through voice output, that comes from the TTS engine discussed earlier. The voice browser hosts the xml files that describe the possibilities of interaction with the user, and a java application that receives the http requests from the browser, according to the xml file that serves the conversation with the user. In that way, the voice browser is the main module of the voice platform, as it is responsible for all the processing being done to the voice that is received from or served to the user. The voice browser also acts as a SIP server, as the user's command line sip phone calls the VoiceXML browser directly without the user being aware of the call.

- **Voice User Interface – GUI synchronization**

  The voice interface is in complete synchronization with the graphical user interface, in order for the user to be able to switch back and forth from one to another, or use them in parallel. When the user executes a command by voice, the interface makes an http request to the **Interaction Manager**, to inform it about the action that took place. The GUI in turn, changes state as if the user has performed the same action using the web interface. In the same manner, when the user performs an action using the web interface, if s/he has first enabled the voice interface, the appropriate voice xml file is loaded to the VoiceXML browser, in order for the user to be able to continue his actions using his voice instead of the web interface from this point on. The voice interface is not enabled by default, and the user must enable it if he wants to start using his voice instead of the web.

7.4 Examples of use cases mapped to the platform modules

- **Search users**

  Platform modules: Graphical/Voice User Interface for the interaction with the user, the PeerAssist interaction manager, the SLA and the Central Matching System for the matching of the users with the constraints.

- **Do monitored tasks**

  Platform modules: The Graphical/Voice User interface along with the PeerAssist Interaction Manager for the interaction with the user, the IP camera, the microphones and in some cases the motion and door sensors.

- **Raise an alarm**

  Platform modules: The panic button, the SLA and the PA will select the users to be informed about the alert raised from the user.

- **Get help from Personal Assistant**

  Platform modules: The Graphical/Voice User Interface and the Personal Assistant, that will show the user what to do or take control and handle all the work, giving feedback to the user after completing the work.

8 Conclusions

In this deliverable we have identified all major components of the PeerAssist architecture and clarified their role and interactions. We have also identified all major sub-components of each major component. The activities of the main PeerAssist use cases and how they are mapped to the major components is also discussed extensively in Appendix A. Furthermore, in Appendix B, we define the interfaces exposed by each component and their main methods.

This document can serve as a concrete specification for the implementation phase of the project and facilitate the parallel development of components and their subsequent integration and testing.

9 Appendix A – UML Activity Diagrams for the Use Cases

The purpose of this appendix is to present a set of UML activity diagrams providing a high-level viewpoint of the most important PeerAssist use cases. These diagrams are intended to help understand the basic tasks that are required for each use case and identify the components in charge of carrying them out. To serve this goal, the diagrams have been kept relatively simple, while abstraction has been applied where it was possible to allow reuse of common activities (e.g. Make Suggestions, Search Item, etc.) in different contexts.

9.1 Create Group

The diagram displays the steps followed in the Create Group activity along with the main components, which are involved in its realization. Each task of the activity is carried out by a particular component, while the connecting arrows indicate the overall control flow. Since the diagram was deliberately kept at a high level of abstraction, no data flows are presented.

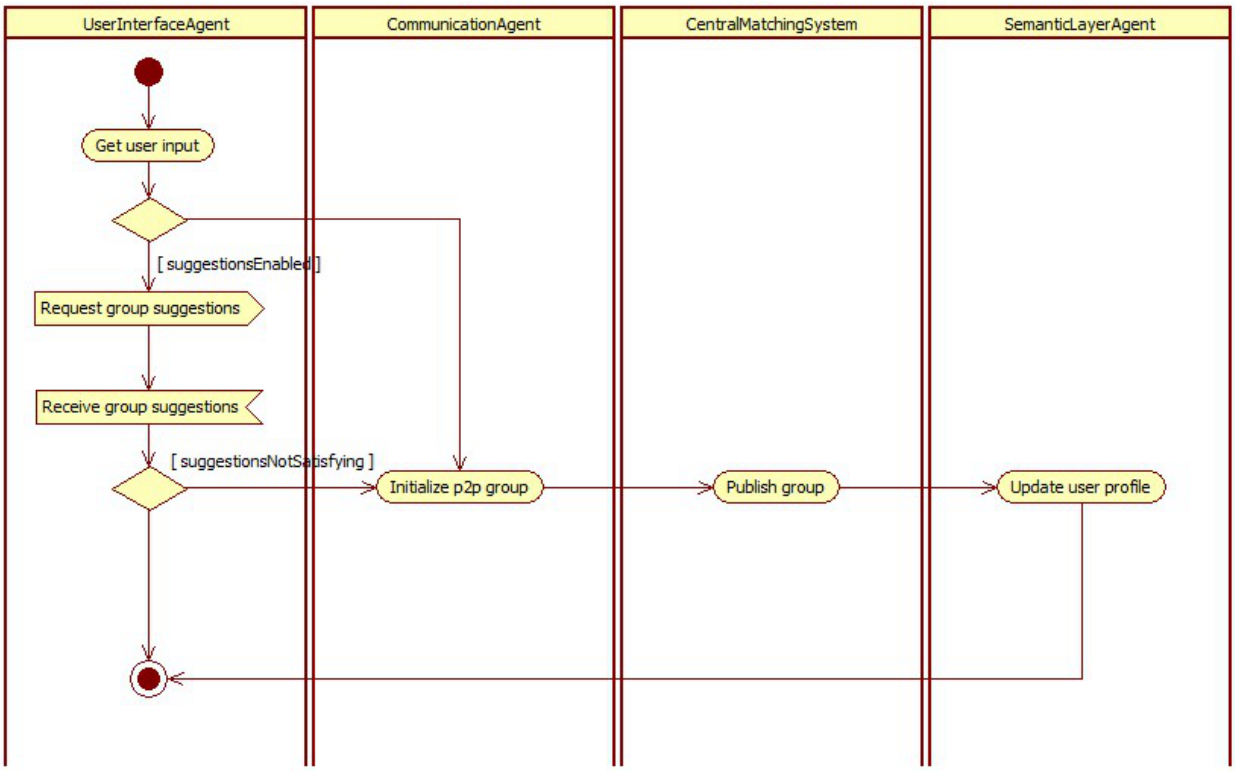

Illustration 1: The "Create Group" use case.

As it can be seen in the figure, the activity is mainly controlled by the User Agent Interface (UAI). The UIA captures the user input and checks whether the user wants to receive suggestions from the Personal Assistant (PA), or not. In

case the user preferences indicate that suggestions are enabled, the UIA asynchronously communicates with the PA in order to receive group-related suggestions.

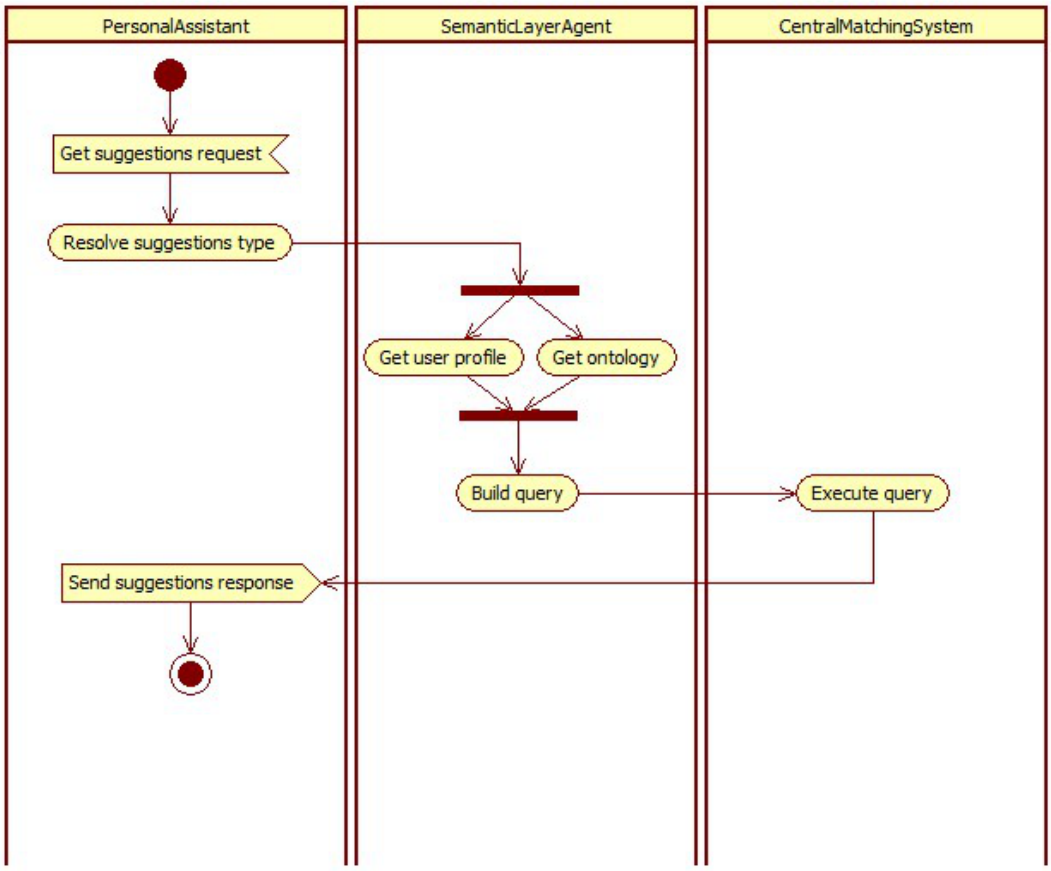

Illustration 2: The "Make Suggestions" use case.

Upon receiving a suggestions request, the PA resolves its type (in this case, the suggestions refer to groups). Based on that information, the SLA retrieves the user profile and the PeerAssist ontology in order to build the appropriate query, which is then executed by the CMS. The matching groups are returned to the PA, which finally sends them back to the caller (in this case, the UIA).

If the suggestions are satisfying, i.e. the user prefers to join an already existing group that is similar to the one she intended to create, the use case is immediately completed. Otherwise, the generation of the new group begins. Firstly, the Communication Agent (CA) initializes the peer-to-peer group performing all necessary network-level tasks. Then, the Central Matching System (CMS) takes over and publishes the generated group advertisement. Finally, the user profile is locally updated by the Semantic Layer Agent (SLA), so as to reflect the user's ownership and participation in the new group.

9.2 Join Group and Leave Group

Provided that the user is granted permission, the process of joining or leaving an existing group is straightforward. The UIA captures the selected group and asks from the CA to perform the necessary steps in order to join/leave the underlying p2p group. As soon as the CA joins/leaves the group, the CMS updates the group advertisement so as to include/remove the established membership, while the SLA updates the user profile in a similar way.

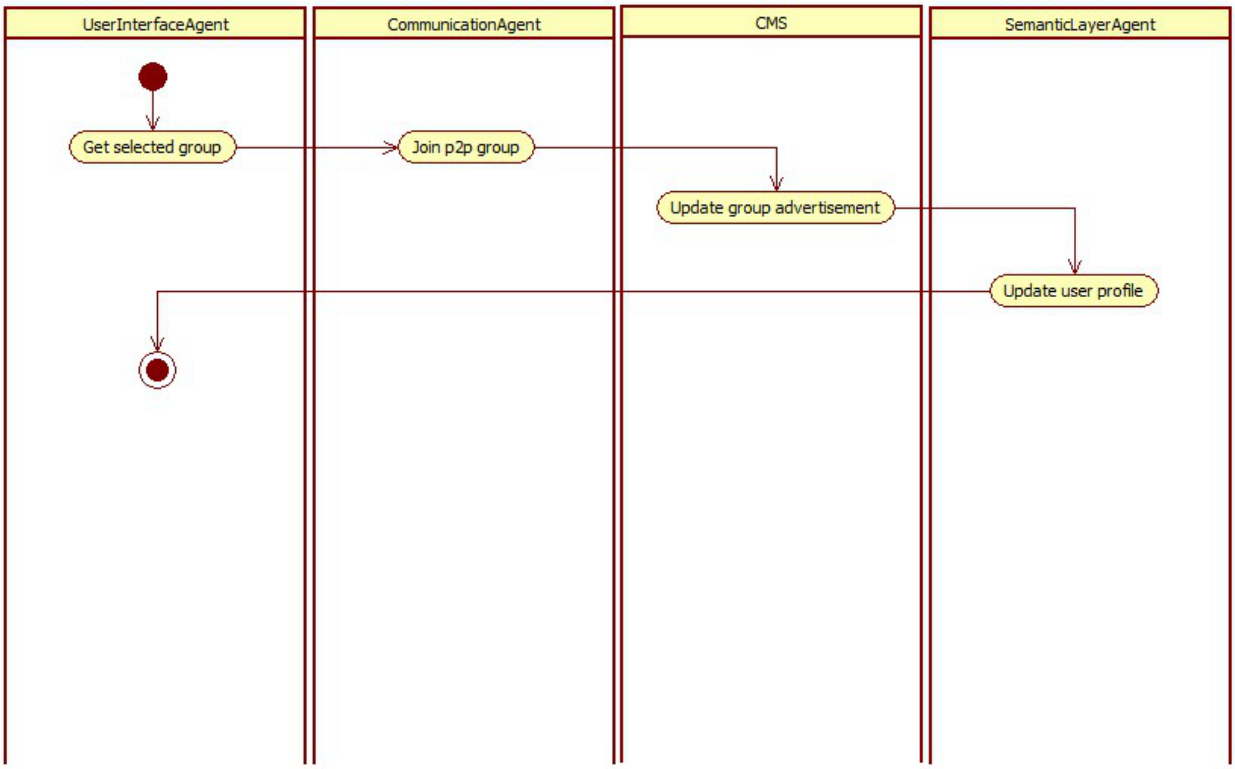

Illustration 3: The "Join Group" use case

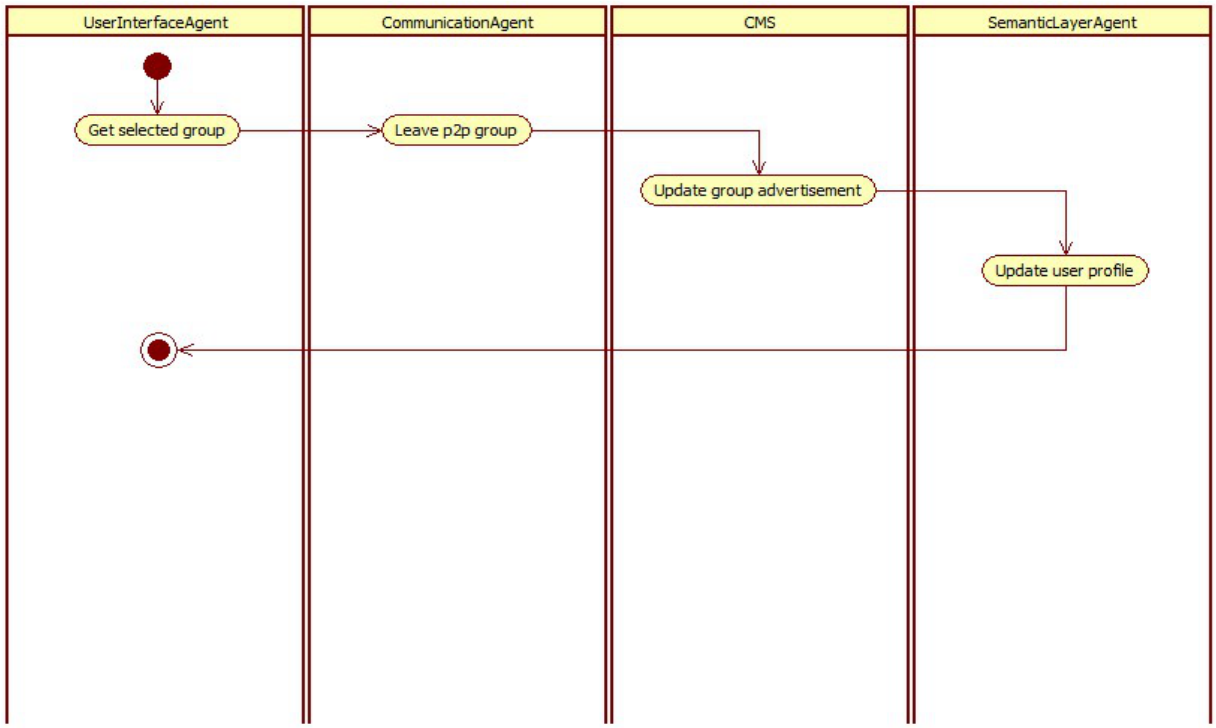

Illustration 4: The "Leave Group" use case

9.3 Delete Group

Deletion of an existing group may be performed only by the group owner, i.e. the user that originally created that group. The UIA captures the user selection and calls the CMS in order to retrieve details about the group to be deleted. Within those details is the list of group members, which is used by the CA in order to notify them that the group will be deleted. It is expected that, all group members respond to that notification by releasing their group-related resources (e.g. communication channels) and appropriately updating their user profile. Nevertheless, that behavior is not part of the use case and therefore it has not been modeled in the activity diagram.

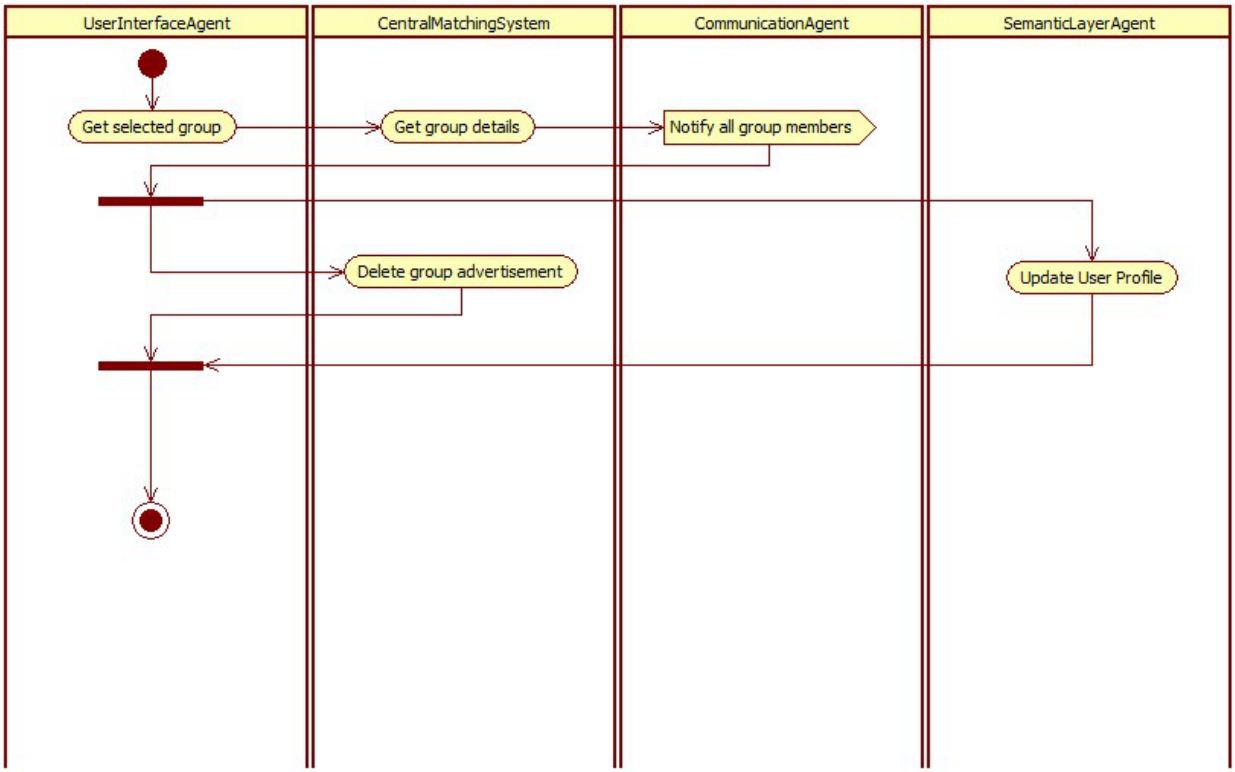

Illustration 5: UML activity diagram for the Delete Group use case

Similarly, as soon as notifications are sent to all group members, the owner user asks from the CMS to delete the corresponding group advertisement, and accordingly updates her user profile. At this point, the activity is complete.

9.4 Invite Users

A user owning a particular group is able to send invitations to other users so that they join. The UIA gets the group selected by the user and checks whether suggestions from the PA are enabled or not. If suggestions are enabled, the UIA sends a request for user suggestions, i.e. for users which might be interested in joining the selected group. If the suggestions given by the PA are not satisfying, the UIA proceeds with a search based on criteria set by the user, executing the Search Item use case.

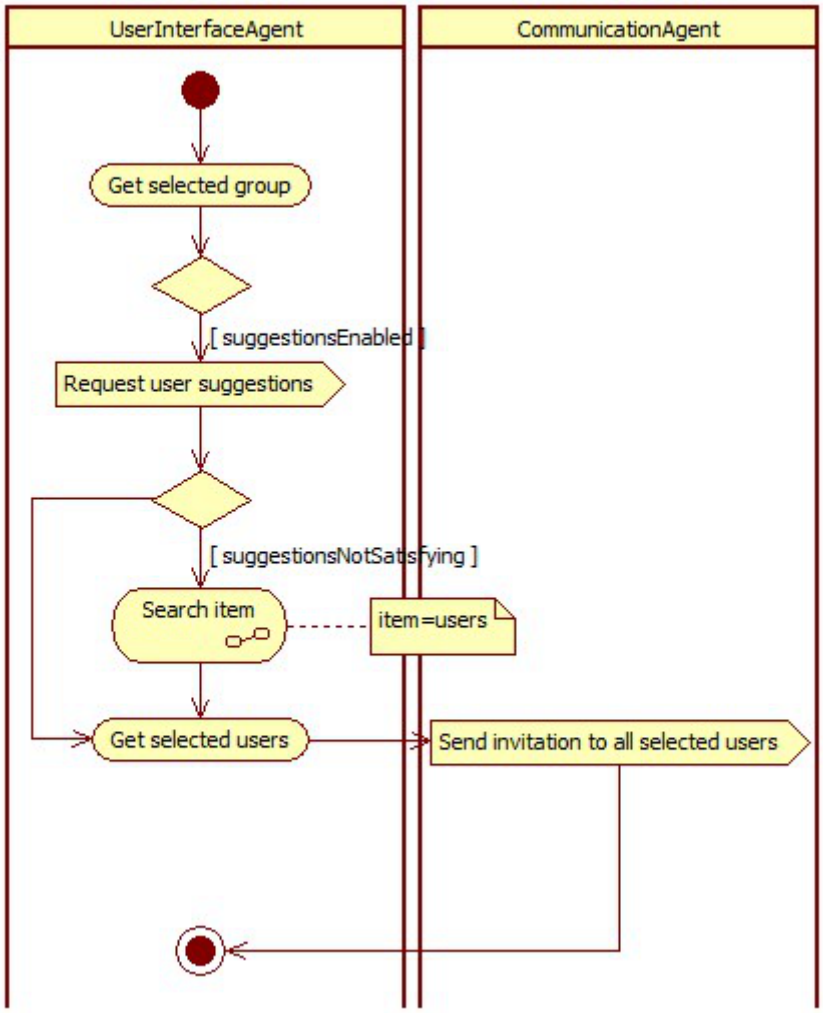

Illustration 6: The "Invite Users" use case

On the basis of the suggestions received by the PA, or the search results, the user is able to select the users she wants to invite. The UIA captures that selection and sends it to the CA, which completes the activity by sending invitations to all selected users.

9.5 Remove User from Group

There may be cases where the owner of a group wants or needs to remove a particular group member. This activity is performed according to the following diagram:

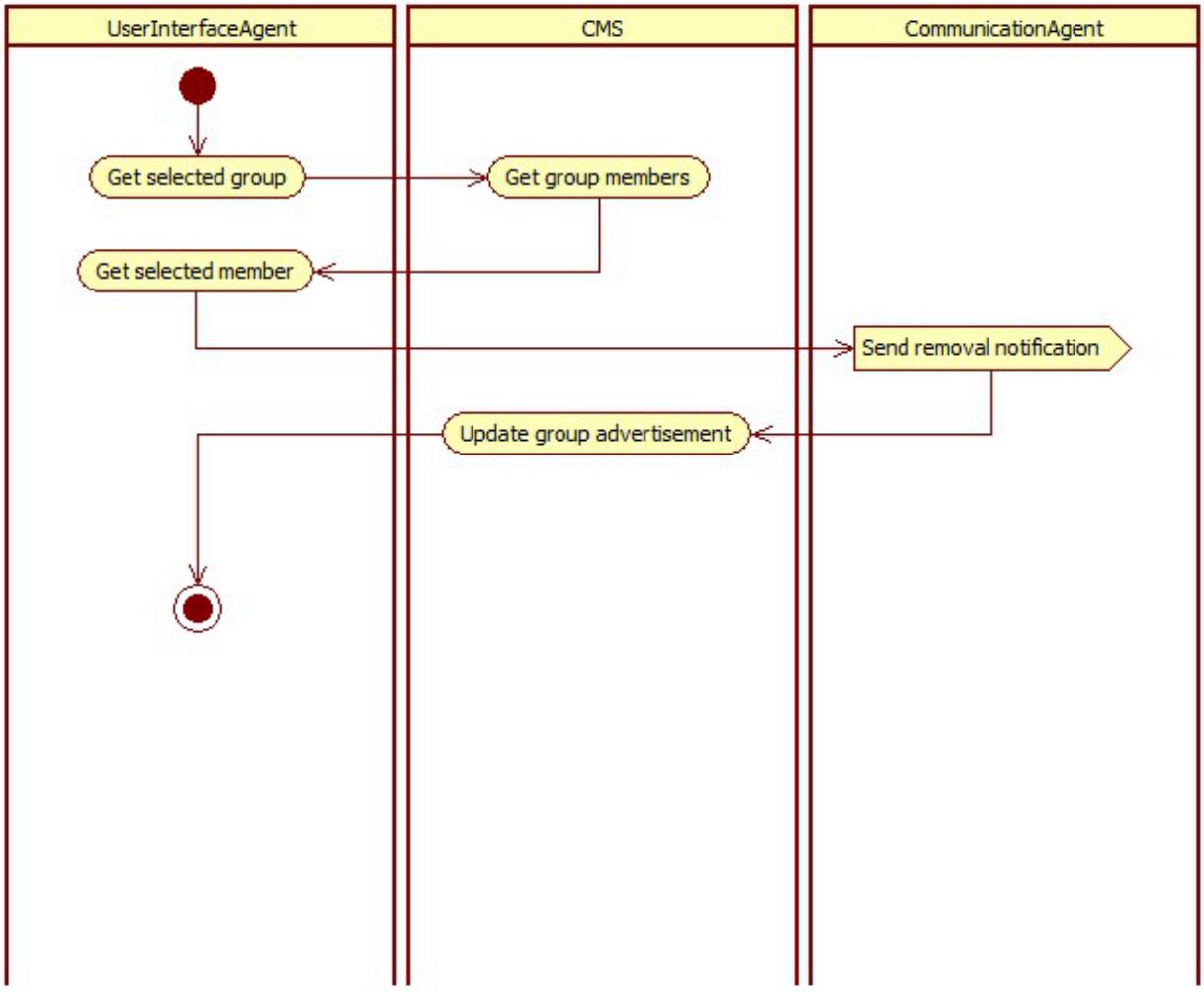

Illustration 7: The "Remove User from Group" use case

Based on the group selected by the user, the UIA asks the CMS for all its current members. Then, the UIA captures the selection made by the user and uses the CA to notify the member of its removal from that group. Finally, the CMS updates the group advertisement so as to reflect the change in the list of group memberships.

9.6 Search Item

The following diagram depicts the tasks and components involved whenever the user needs to search for an item. Within the PeerAssist context, an item can be any kind of resource, e.g. a group, a user, an event, etc.

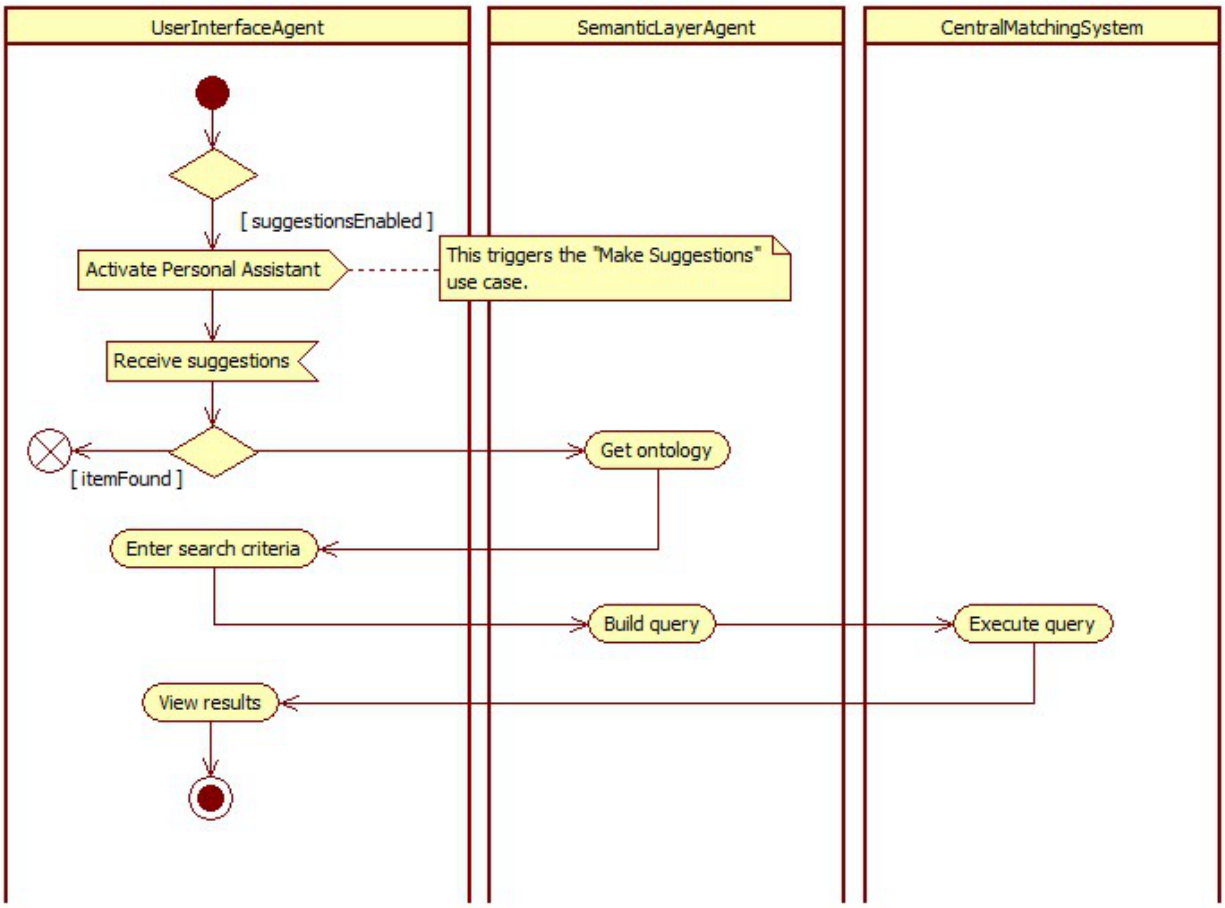

Illustration 8: The "Search Item" generic use case

As it can be seen in the diagram, the search activity triggers the PA, which in turn makes related suggestions to the user according to the Make Suggestions use case. As soon as the UIA receives those suggestions, the user is able to inspect them and decide whether the desired item is found or not. In case the user still wants to perform a search, the UIA retrieves the PeerAssist ontology from the SLA, so that the user is assisted in filling in her search criteria with the use of semantics. The query is built by the SLA and is executed by the CMS, while the search results are sent back to the UIA, marking completion of the search activity.

9.7 Advertise Event

The advertising of an event is straightforward and is shown by the following UML activity diagram.

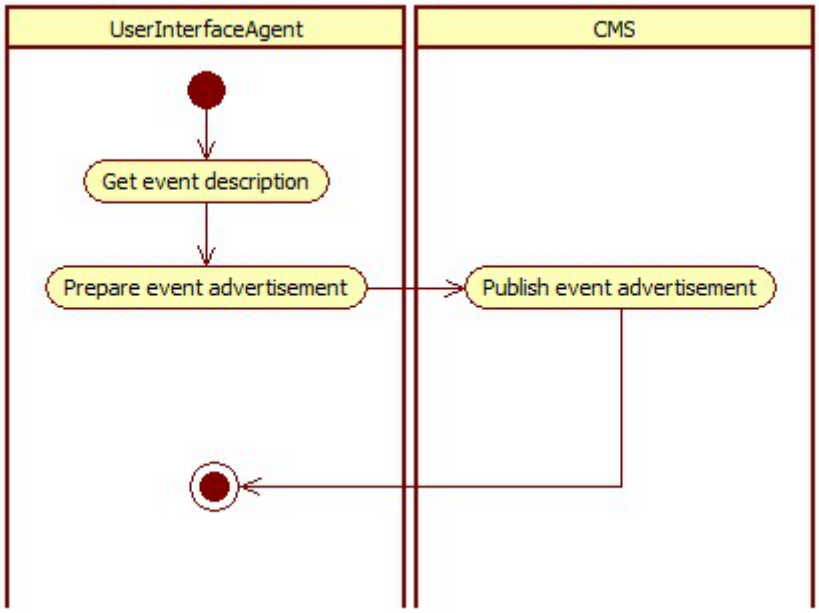

Illustration 9: The "Advertise Event" use case

9.8 Raise Alarm

The purpose of the Raise Alarm activity is to connect the user in need with one of her registered caregivers.

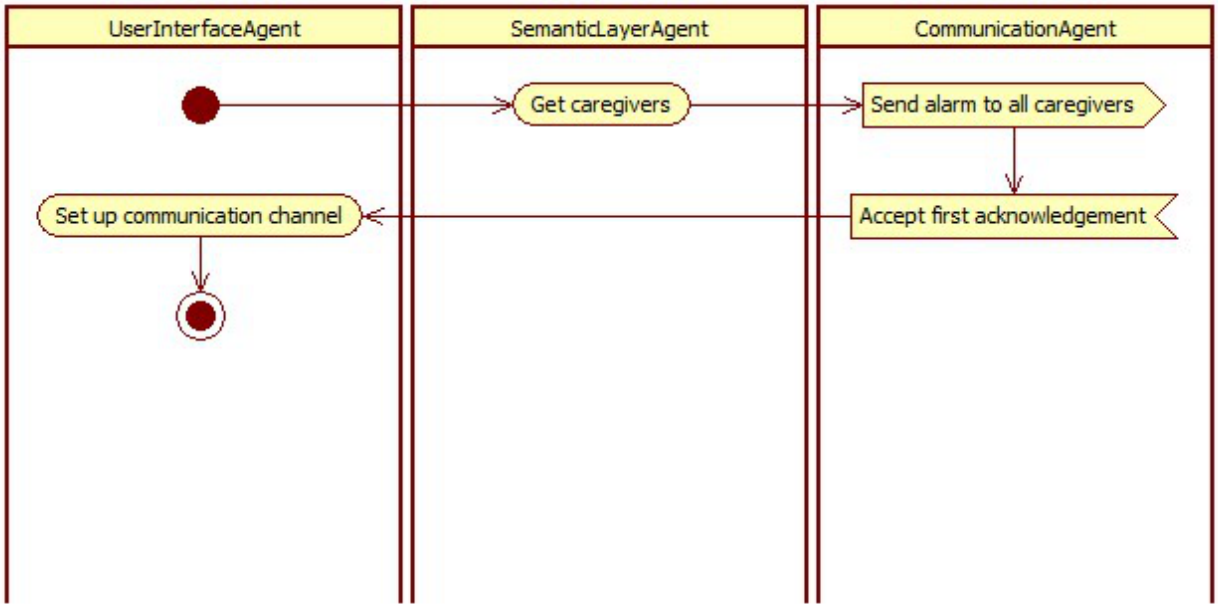

Illustration 10: The "Raise Alarm" use case

Triggered by the UIA, the SLA retrieves the list of registered caregivers and sends it to the CA. In turn, the CA broadcasts an alarm message to all caregivers and awaits for their response. The UIA sets up the appropriate communication channel to allow the user get in contact with the first caregiver that sends back his acknowledgment.

9.9 Organize Event

As it can be seen in the diagram below, the Organize Event activity is modeled as a composition of three other activities, namely the Create Group, Invite Users, and Advertise Event.

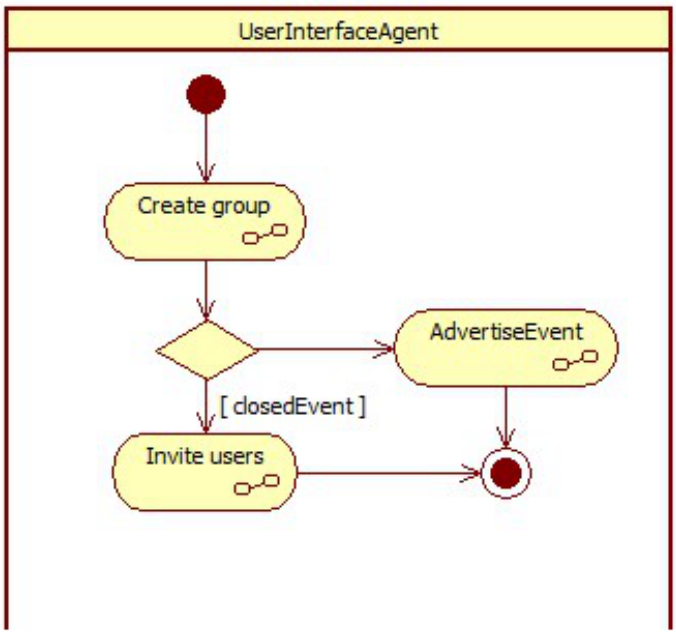

Illustration 11: The "Organize Event" use case

More specifically, the organization of an event by a user involves the following main tasks: First, the user needs to create a group that will be dedicated and associated to the event. If the event is not closed, the user proceeds with its advertising so as to render it discoverable by others. Otherwise, the user launches the Invite Users activity in order to populate the group with the desired members, which will participate to the event.

9.10 Consult Doctor

The activity of consulting a doctor comprises a set of tasks as shown in the following UML activity diagram.

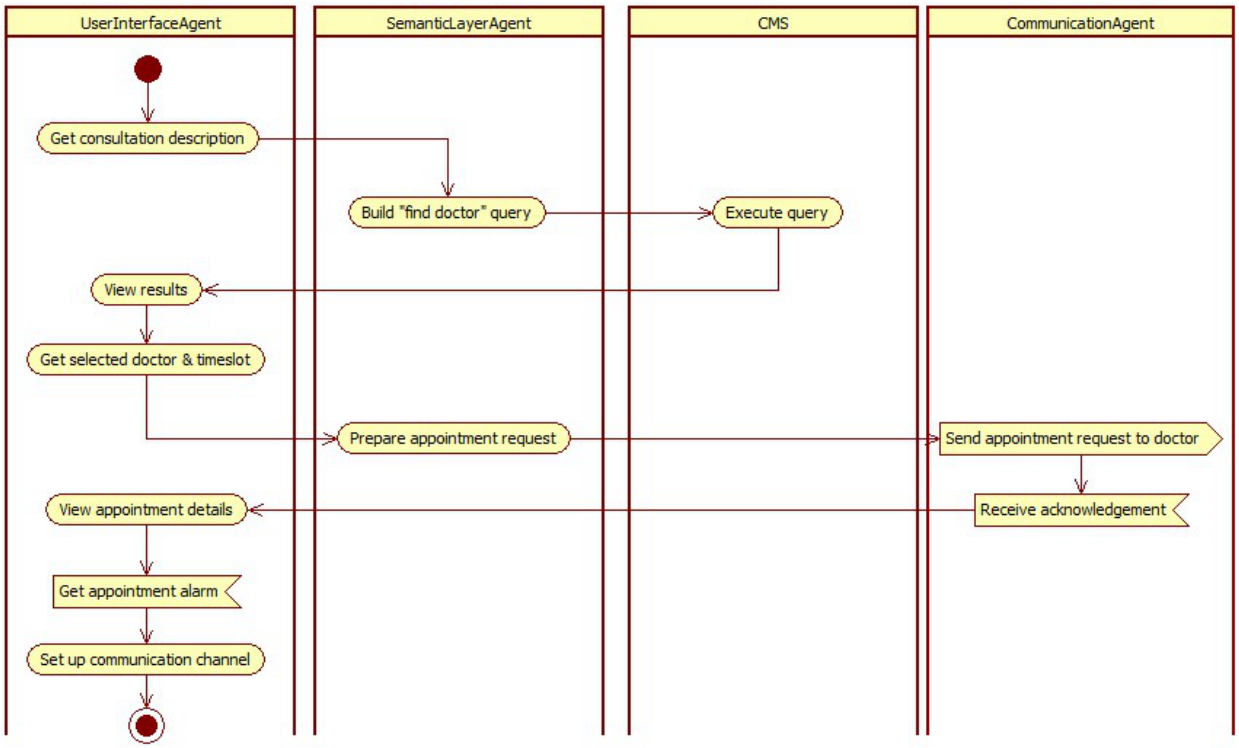

Illustration 12: The "Consult Doctor" use case

Based on the description given by the user, the SLA builds the appropriate doctor query, which is then executed by the CMS. The results are viewed by the user, who is able to select a doctor and available time slot. Based on that selection, the SLA prepares an appointment request, which is sent to the doctor through the CA. As soon as the latter receives the doctor's acknowledgment of availability, the appointment is considered confirmed. At this point, the user may wait until an alarm is triggered by the UIA to inform her of the appointment. Upon this alarm, the appropriate communication channel is set up by the UIA in order to realize the appointment and allow the user consult the selected doctor.

10 Appendix B – Defined Interfaces

From PA to UIA – Interface implemented by UIA

notify(ntfObject): void

Description: Show some sign on screen, possibly interrupting the user if critical. ntfObject stored at the ontologies consists of 1) ntfId 2) ntfType (alarm, invitation, etc.) 3) ntfOriginator (userId) 4) timeStamp 5) groupId (only for invitations) 6) ntfReadflag 7) ntfMessage (optional by the originator)

Comments: E.g. invitation received, request to enter a conference with a doctor, medication reminder, emergency detected by sensors, etc.

userChatMessageReceived(userId, message): void

Description: The PA informs the UIA for a user chat message that has just been received, sent by user "userId".

Comments:

groupChatMessageReceived(userId, groupId, message): void

Description: The PA informs the UIA for a group chat message that has just been received, sent by user "userId" to group "groupId".

Comments:

From PA to CA – Interface implemented by CA

inviteUsers(groupId, userList, ntfMessage): void

Description: Invite a list of users to join a group (sent by the group owner).

Comments: The userList is the list of userIds that are to be invited to the group with groupId. In an invitation scenario, the group owner (with userId) has to invite a list of users to a specific group. This userList has to be defined by the user, thus, the user has first to retrieve such users. The group owner can send an optional message together with the invitation. The CA adds the timestamp before sending this message out.

joinGroup(groupId, userId): void

Description: Join a group as a response to an invitation (for closed groups) or not (for open groups).

Comments: The user with userId wants to join a group. This is clear.

createGroup(groupId): void

Description: Create a new group.

Comments: This implies that the PA knows the groupId as provided by the SLA (createGroup towards SLA is performed first, to get the groupId, followed by createGroup towards the CA).

deleteGroup(groupId): void

Description: Delete a group (only by the group owner)

Comments: This is clear, assuming again the PA knows the groupId. If the UIA provides only the name, it may need to access the SLA to get the groupId.

leaveGroup(groupId, userId): void

Description: The user leaves a specific group or the group owner removes a user from group

Comments: This is clear, provided Ids are known.

notifyGroup(ntfType, userId, groupId, ntfMessage): void

Description: Send a message to a specific group (e.g. used in the Raised Alarm Scenario)

Comments: Once the message refers to the panic scenario situation then the groupId in the parameter refers to the pre-defined groups of the Caregivers. In case of motion (non)detection event, the groupId refers to the friends and family group. The CA adds the timestamp before sending this message out.

userChatMessage(userId, message): void

Description: Sends a chat message to another user.

Comments: To clarify if CA can send direct messages to a specific user.

groupChatMessage(groupId, message): void

Description: Sends a chat message to a group.

Comments: Similar as the previous one.

From UIA to PA – Interface implemented by PA

createUser(userInfo): userId

Description: (relay to SLA) Creates a new user and gets the userId.

Comments: A question is if more than one users can be connected through the same PeerAssist terminal. If not, this method should be available only once and then only choose updateUserInfo() to change the user information. If yes, then we have to add a "userId" parameter in some

methods below to say who is the user that calls them. Also the SLA should have provision for more than one users. Left for next version.

updateUserInfo(userId, userInfo): void

Description: (relay to SLA) Updates the information on a user.

Comments: Same as before, to include the userId depends on the number of users per terminal.

getQueryTemplate(templateType): queryTemplate

Description: (relay to SLA) Request from SLA a query template of the given type in order to generate a form on the UI.

Comments: Parameters are undefined.

getSuggestions(suggestionsType): itemList

Description: (relay to SLA) Request items of the given type (users/groups/services) that match the user.

Comments: Parameters are undefined.

launchQuery(query): itemList

Description: (relay to SLA) Send a query to retrieve users/groups/services.

Comments: Parameters are undefined. The query data type has to be specified at least in this version. Depends on the structure of the ontologies and the UIA needs for information in all scenarios.

getUserGroups(): groupList

Description: (relay to SLA) Get the groups the user is currently a member of.

Comments: It is implied that the groupList is a set of the groupIds. Otherwise this list has to be defined. Couldn't that ne implemented through the launchQuery method with the proper query parameter?

getUserInfo(userId): userInfo

Description: (relay to SLA) Get a user's info. If it is the same user, the full user profile is returned, otherwise the public profile is returned (i.e., null values in private fields).

Comments: Once the user info is stable, this is clear.

getGroupInfo(groupId): groupInfo

Description: (relay to SLA) Get detailed info about the group.

Comments: Needs detailed description of what the groupInfo should be.

getEventInfo(eventId): eventInfo

Description: (relay to SLA) Get detailed info about the event.

Comments: Needs detailed description of what the eventInfo should be.

getActivityInfo(activityId): activityInfo

Description: (relay to SLA) Get detailed info about the activity.

Comments: Needs detailed description of what the activityInfo should be.

createGroup(groupInfo): groupId

Description: Register new group in SLA, then create group on CA. The groupId as returned by the SLA to PA, is provided to UIA, and used to create a group on the CA.

Comments: It should be clarified what the groupInfo is.

joinGroup(groupId): void

Description: Join group on CA, register that on SLA.

Comments: This is clear, assuming the UIA knows the groupId. If groupId is the group name this is OK.

leaveGroup(userId, groupId): void

Description: This method is called either by a user to leave a group, or by the group owner to delete a user from a group.

Comments: When the group owner wants to delete a user from the group, (s)he can use leaveGroup with the specific user's userId, while when any user wants to leave a group, (s)he can use again leaveGroup with his/her userId.

deleteGroup(groupId): void

Description: Delete group on CA, register that on SLA.

Comments: This is clear. Only the group owner can perform this task.

getGroupMembers(groupId): userList

Description: (relay to SLA) Send a query to retrieve all users of a group.

Comments: This method has to return a userList. This includes only the userIds or something more like the user name?

inviteMatches(groupId): userList

Description: (relay to SLA) PA requests from SLA a userList matching the given group and returns that to UIA, to allow the user selecting where to send invitations.

Comments: This is clear

inviteUsers(groupId, userList, ntfMessage): void

Description: PA sends invitations through CA to the given userList.

Comments: This is clear.

getNotifications(ntfType): listOf(ntfId, userId)

Description: (relay to SLA) Get the queue of unread notifications.

Comments: This is clear but implies that the SLA stores somehow the message sequence of the notifications.

userChatMessage(userId, message): void

Description: (relay to CA) Sends a message to another user.

Comments: To clarify if CA can send direct messages to a specific user.

groupChatMessage(groupId, message): void

Description: (relay to CA) Sends a message to a group.

Comments: Same comment as before

From HAC to PA – Interface implemented by PA

trigger(triggerId): void

Description: The Home Automation Controller triggers the PA once an event is detected.

Comments: The type of the event is described in the triggerId parameter. At the time being, the triggered can be either motion-detection event and panic-button-pressed event. The HAC can define more types of events in the future, and the PA can handle them. Such information should trigger notifications through the CA and the corresponding UIA in order to notify the pre-defined groups of the user and the display of the user respectively.

From CA to PA – Interface implemented by PA

notificationReceived(userId, ntfType, timeStamp, groupId, ntfMessage): void

Description: The CA receives a notification through JXTA (e.g. alarm or invitation) and forwards it upwards.

Comments: The complete cycle of notifications should be described.

userChatMessageReceived(userId, message): void

Description: The CA informs the PA for a user chat message that has just been received, sent by user "userId".

Comments:

groupChatMessageReceived(userId, groupId, message): void

Description: The CA informs the PA for a group chat message that has just been received, sent by user "userId" for group "groupId".

Comments:

From PA to SLA – Interface implemented by SLA

createUser(userInfo): userId

Description: Creates a new user to SLA/CMS and gets the userId.

Comments: A question is if more than one users can be connected through the same PeerAssist terminal. If not, this method should be available only once and then only choose updateUserInfo() to change the user information. If yes, then we have to add a "userId" parameter in some methods below to say who is the user that calls them. Also the SLA should have provision for more than one users. Left for later.

updateUserInfo(userId, userInfo): void

Description: Updates the information on a user.

Comments: Same as before, to include the userId depends on the number of users per terminal. Left for the future.

getQueryTemplate(templateType): queryTemplate

Description: Returns the query template associated with a certain template type.

Comments: The actual template types have yet to be defined.

getSuggestions(userId, suggestionsType): itemList

Description: Returns suggestions for a specific user. itemList should be adjusted accordingly.

Comments: Suggestion types include: Group, Activity, Event and User.

launchQuery(query): itemList

Description: Launches a query which is redirected from SLA to CMS if necessary.

Comments: What will launch query return without having a query type as parameter? Probably depends on the query.

getUserGroups(userId): groupList

Description: Returns all groups which a user is member of.

Comments: Also probably later introduce this for Activities and Events. The groupList contains only groupIds are more group info (like group names for example)?

getUserInfo(userId): userInfo

Description: Get a user's info. If it is the same user, the full user profile is returned, otherwise the public profile is returned (i.e., null values in private fields).

Comments: Once the user info is stable, this is clear.

getGroupInfo(groupId): groupInfo

Description: Returns detailed info about the group.

Comments: Needs detailed description of what the groupInfo should be.

getEventInfo(eventId): eventInfo

Description: Returns detailed info about the event.

Comments: Needs detailed description of what the eventInfo should be.

getActivityInfo(activityId): activityInfo

Description: Returns detailed info about the activity.

Comments: Needs detailed description of what the activityInfo should be.

createGroup(userId, groupInfo): groupId

Description: Creates a group with group information.

Comments: Should be checked if the userId is needed, or the SLA already has the userId. The user who is the creator of the group will be responsible to manage Group memberships and check added Activities, Events and Topics.

joinGroup(userId, groupId): void

Description: User joins a group.

Comments: There will be two types of groups, open and closed. The user can freely join open groups, but needs an invitation for closed groups. In any case, the SLA method is called by the PA only after the user has successfully joined the group at JXTA level.

leaveGroup(userId, groupId): void

Description: Called by a user to either leave a group or by the group owner to delete another user from the group.

Comments:

deleteGroup(userId, groupId): void

Description: Deletes a group from SLA/CMS.

Comments: Only the group creator can delete the group.

getGroupMembers(groupId): userList

Description: Returns a list of the group members.

Comments: This method has to return a userList. This includes only the userIds or something more like the user name?

inviteMatches(groupId): userList

Description: Returns a list of users that match a certain group profile.

Comments: Same comment as before.

getNotifications(ntfType): listOf(ntfId, userId)

Description: Get unread notifications of the specific type

Comments: Again, are the IDs enough or we need more (ntfTypes and user names)?

getNotification(ntfId): ntfObject

Description: Get full record of the specific notification.

Comments: Clear.

notificationReceived(userId, ntfType, timeStamp, groupId, ntfMessage): void

Description: Provide the notification details to be stored in SLA.

Comments: Clear.