D4.2 Mobile Notification System

weSPOT

IST (FP7/2007-2013) under grant agreement N° 318499

Deliverable number	*D4.2*
Deliverable Title	*Mobile Notification System*
Dissemination level	*PU*
Delivery date	*31 March 2014*
Status	*Final*
Author(s)	*Stefaan Ternier*
Contributors	*Angel Suarez, Marcus Specht*

The weSPOT Consortium

Number	Partner	Country
1	OPEN UNIVERSITEIT NEDERLAND	Netherlands
2	THE OPEN UNIVERSITY	United Kingdom
3	TECHNISCHE UNIVERSITAET GRAZ	Austria
4	KATHOLIEKE UNIVERSITEIT LEUVEN	Belgium
5	SOFIISKI UNVERSITET SVETI KLIMENT OHRIDSKI	Bulgaria
6	FRIEDRICH-ALEXANDER-UNIVERSITAT ERLANGEN NURNBERG	Germany
7	FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS	Greece
8	LATTANZIO LEARNING SPA	Italy
9	IPAK INSTITUT ZA SIMBOLNO ANALIZO IN RAZVOJ INFORMACIJSKIH TEHNOLOGIJ VENENJE ZAVOD	Slovenia

Document Control

Coordinating Editor: *Stefaan Ternier*

E-mail: **stefaan.ternier@ou.nl**

Amendment History

Version	Date	Author/Editor	Reviewers	Description/Comments
1	26/02/2014	Marcus Specht		Alfa version
2	11/03/2014	Stefaan Ternier, Angel Suarez		Initial contributions on MNS
3	26/03/2014		Atanas Georgiev, Elisabetta Parodi	Draft for internal review
4	31/03/2014	Stefaan Ternier		Finalised text incorporating feedback from the internal review

1. Executive Abstract

This deliverable presents the Mobile Notification system. It reports on the outcomes of the work conducted so far under Task 4.2: Context-aware notification system.

In Part A of the DoW, this task is described as:

"The mobile contextual notification system will enable the notification of mobile users based on open notification protocols as XMPP. Using sensor technology in smartphones the notification of users can be based on contextual filters relevant for inquiry projects. Learners can link inquiry projects to certain locations, physical objects, or combinations of contextual factors, i.e. the weather at a certain location at a specific time of the year. Furthermore via notifications a shared inquiry workflow can trigger the collection of data dependent on several parameters (location, time, social context, environment). The system can also be used for ubiquitous and context-dependent messaging services integrated with social media for mobile notification and communication related to inquiry projects."

The present document describes a framework for managing mobile notifications that makes a clear distinction between context and notifications.

Table of Contents

1. Introduction

Several components in the weSPOT architecture require learners to be notified. Conceptually, a notification refers to the action of informing the user of something. A notification system will therefore facilitate this process of informing the user that an event has occurred. In this deliverable, a distinction will be made between the system that processes and delivers the notification to the user and the event for which a notification was sent.

The weSPOT Mobile Notification System (MNS) was developed as a software component that enables the various weSPOT components to send notifications to the learner or teacher regardless of the technology these components use (Android SDK, iOS SDK, web application). These systems remain however unaware of the type of event that the notification is sent for. An event can be, for example, the availability of a new data collection task or the arrival of a new message.

A notification is triggered by a system event (e.g. new message available). After receiving a notification, the client system will synchronize with the server (download the message) and update the user interface (UI), making the user aware of this system event. As a result, the user will open the message for which a notification was sent.

In a web service based architecture like the weSPOT architecture (Figure 1), client applications pull data via web services from the server (e.g. the ARLearn cloud (Ternier, 2012), or the Inquiry Workflow Engine (Mikroyannidis, 2013)). Client applications however do not implement web services that would enable cloud applications to contact them. In order to implement a bidirectional communication channel that enables the server to communicate with the client, the weSPOT notification system builds upon existing technologies.

1. The Google Channel API creates a persistent connection between the client and server. This technology offers a JavaScript client that delivers notification messages to the client without the use of polling. A long-held HTTP request enables the server to push data to the HTTP client. As a notification message is available for the client, the server will return an HTTP response to the long-held HTTP request.
2. Google Cloud Messaging (GCM) helps developers to send data from a server to an Android application. Rather than having every single Android application maintaining an open bidirectional communication channel with the server, this technology enables sharing a persistent XMPP communication channel across applications.
3. Apple Push Notifications (APN) is a technology similar to GCM that enables asynchronously sending a notification to an iOS device. The payload of an APN JSON message is constrained to 256 bytes. APN messages are thus suited for “send-to-sync” messages.

In addition to a notification engine, this deliverables also contributes a messaging system. This system enables inquiry users to communicate and exchange messages. Chapter 2 describes the MNS architecture and positions it in the weSPOT architecture. This chapter illustrates how notifications enable data collection tasks and messaging between the weSPOT components. Chapter 3 applies the MNS architecture to three weSPOT components.

The appendices of this deliverable are devoted to the API’s. The MNS API (Appendix 1) is a generic API for sending notifications regardless of their use. This API can be used by any third party application to send notifications to the weSPOT components. The Messages API (Appendix 2) provides a high level API for managing messages. With this API both threads and messages are created. Behind the curtains, this API builds on the MNS API to notify the user’s device.

2. Architecture

The MNS has been implemented as an ARLearn component providing an API that enables access for third party applications. Figure 1 presents the entire weSPOT architecture and positions the MNS as an ARLearn cloud component. All components relevant for this document are highlighted in this figure.

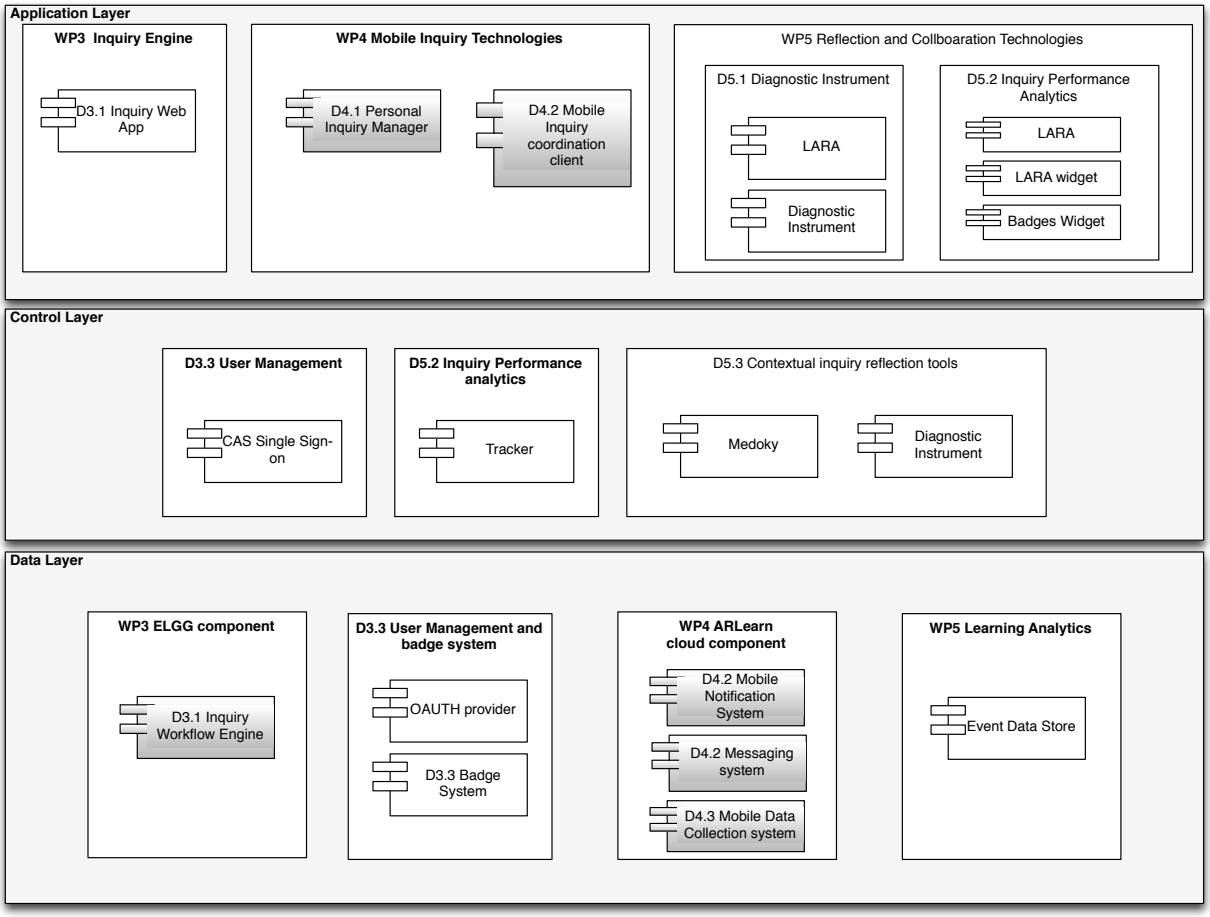

Figure 1: The Mobile Notification System in the overall weSPOT architecture.

The Mobile Notification System (MNS) can be used in two ways:

1. As a standalone notification system, it offers an abstraction layer on top of technologies such as GCM, APN and the Google Channel API and enables third party applications to broadcast notifications.
2. Embedded in the WP4 ARLearn cloud components. All ARLearn cloud components by default publish every user event via the MNS. When a user creates a new message via the messaging API, a notification will be sent to all recipient devices. When a new data collection task is created, a notification is sent to all devices that participate in this inquiry. In this scenario, the notification is a "tickle" that informs the device that new content is available via respectively the Messages API and the Mobile Data Collection API.

Although both options are currently in production, none of the (non-ARLearn based) weSPOT components make use of the first option yet. Chapter 3 will illustrate how the second option integrates with the PIM, MICI and the chat widget.

3. MNS clients

3.1. Personal Inquiry Manager (PIM)

The Personal Inquiry Manager (PIM) (Ternier, 2013) is the main mobile client component of weSPOT toolset that allows users to get an overview of their current active inquiries and to create new inquiries. The remainder of this section revisits the data collection infrastructures and illustrates how the MNS supports this process.

The architecture shown in Figure 2 illustrates how the various weSPOT components are involved in creating a data collection task.

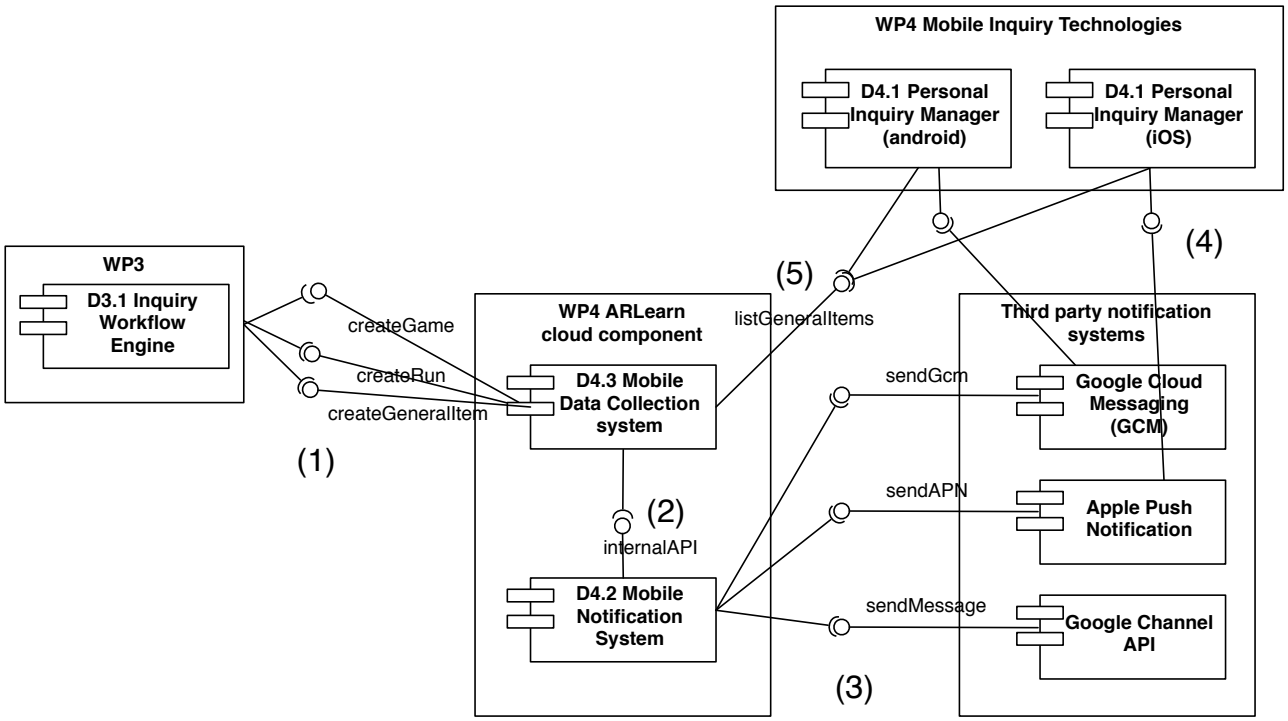

Figure 2: Interaction between MNS and PIM

The user stories that are represented in Figure 2 are "As a teacher, I must be able to create a data collection task" and "As a learner I must be able to execute data collection tasks with the PIM".

When a user creates data collection tasks with the IWE, the IWE creates a generalItem in ARLearn (1). A generalItem is the ARLearn terminology that corresponds to a data collection task. An ARLearn game and run are created once. Every weSPOT inquiry (IWE) has a corresponding ARLearn game and run. Within an ARLearn game an arbitrary amount of data collection tasks can be created. ARLearn communicates data collection updates to the MNS via the internal API (2). As a result, the MNS will iterate over all users that are registered to the inquiry. For each user, it lists the devices registered and sends a notification (3). For iOS devices, the sendAPN message is used to communicate this message over the Apple APN protocol. For Android devices, the sendGCM message is used to update the Android device of the GCM protocol. Note that both Google Cloud Messaging (GCM) and Apple Push Notification (APN) are components that are managed by Google and Apple respectively. Although it is listed in Figure 2, the Google Channel API is not used here. Data collection task notifications are only sent to mobile (PIM) devices and not to web based components.

Next, as the iOS and Android PIM receive an APN or GCM "send-to-sync" notification (4), they will update their list of data collection tasks via the listGeneralItems web service that is offered by the Mobile Data Collection cloud component (5).

Data collection tasks were already introduced in D4.1 and will be presented in further detail in D4.3 (September 2014). The ARLearn framework offers functionality to manage data collection tasks that are created via the IWE. As a teacher creates data collection task via the corresponding IWE widget, the MNS informs all devices that are registered to the inquiry with a "send-to-sync" message.

In addition to this notification mechanism, data collection task have also been made context aware. A data collection task defines a simple lifecycle with three states.

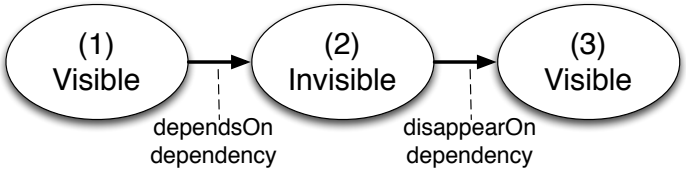

Figure 3: Data Collection task lifecycle

A data collection task can migrate from state (1) to state (2) via a dependency and can next travel to state (3) via another dependency. Tasks can never move to a lower state.

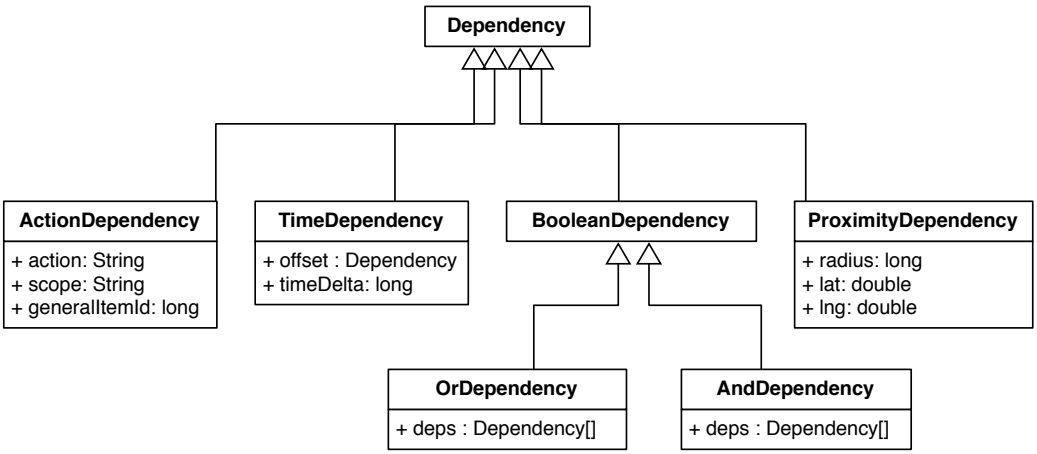

Figure 4: Dependencies define context parameters

Dependencies define context parameters that must be fulfilled in order to move to a higher state. Figure 4 illustrates the dependency model that supports four kinds of dependencies:

1. An action based dependency evaluates to the time at which an action or event is triggered in the PIM. An action-based dependency can for instance fire when a user has completed a data collection task.
2. A time based dependency binds a time offset to another dependency. This dependency is fulfilled when two conditions are met. The nested dependency must have fired and the amount seconds specified by the offset attribute must have elapsed since this moment.
3. Boolean dependencies provide a means to create "AND" and "OR" statements with other dependencies. An AndDependency will evaluate to the highest time of its nested dependencies, an OrDependency will evaluate to the lowest time. An OrDependency will thus fire at the

moment that one of its children has fired. An AndDependency will fire when all of its children have fired.

4. A Proximity dependency fires at the moment the user steps into the defined zone.

For instance, one can specify through a dependency that captures a user entering a location. The following json excerpt illustrates how such a data collection task is specified:

```
{
    "id": 4511825563484160,
    "name": "collect temperature",
    "description": "Take 5 temperature samples of ...",
    "lng": 5.7252906999999595,
    "lat": 51.0341886,
    "dependsOn": {
         "type":
"org.celstec.arlearn2.beans.dependencies.ProximityDependency",
         "radius": 50,
         "lat": 50.98234344363075,
         "lng": 5.8131813249999595
    },
    "disappearOn": { …}
}
```

The context parameters in the dependsOn key defines when a task can move from state (1) to (2) (Figure 3). The disappearOn defines when a task moves from state (2) to (3). In the example above, the dependsOn key specifies a ProximityDependency with a location and radius. A user that enters this zone will receive a notification on the device, indicating the availability of a data collection task. Note that the notification engine will inform the PIM of the existence of this data creation task as soon as it is created. When the context parameters (ie. location in this example) are met, the user will be informed through a notification on the smartphone. This enables these context-aware notifications to work even when no network connection is available at this location, as the PIM can synchronize all data collection tasks for offline use.

The following Boolean dependency structure illustrates "gluing" together dependencies. The "dependsOn" key specifies that the item will appear, when a user has checked in on the given location and 5 minutes have passed since he has given collected data for another task.

```
   "dependsOn": {
         "type": "[...].AndDependency",
         "dependencies": [
              {
                   "type": "[...].ProximityDependency",
                   "radius": 50,
                   "lat": 50.90281327767002,
                   "lng": 5.945118262500046
              },
              {
                   "type": "[...].TimeDependency",
                   "timeDelta": 300000,
                   "offset": {
                        "type": "[...].ActionDependency",
                        "action": "answer_given",
                        "generalItemId": 19766002
                   }
              }
         ]
```

```
}
```

3.2. Messaging between MICI and the PIM

Via the Mobile Inquiry Coordination Interface (MICI) client, teachers are able to monitor progress made within an inquiry. With this tool, they are able to send live instruction to learners using the PIM, while the inquiry is active. This approach contrasts with data collection tasks that are typically defined when the inquiry is being designed.

The PIM allows learners to communicate with each other and with Mobile Inquiry Coordination Interface (MICI). It was decided not to make use of existing - traditional- communication channels such as WhatsApp, Telegram, Google Hangout as these require learners to share their personal accounts with their teacher or peer learners. Rather embedding communication facilities in the scope of an inquiry enables both parties to exchange messages without the burden of setting up the communication infrastructure. From within an inquiry users can send messages either to a single peer or to all participants of the inquiry. The MNS supports this message infrastructure. As a new message is submitted to a thread, all devices will receive a send-to-sync message that enables the PIM to (1) retrieve the message and (2) notify the user.

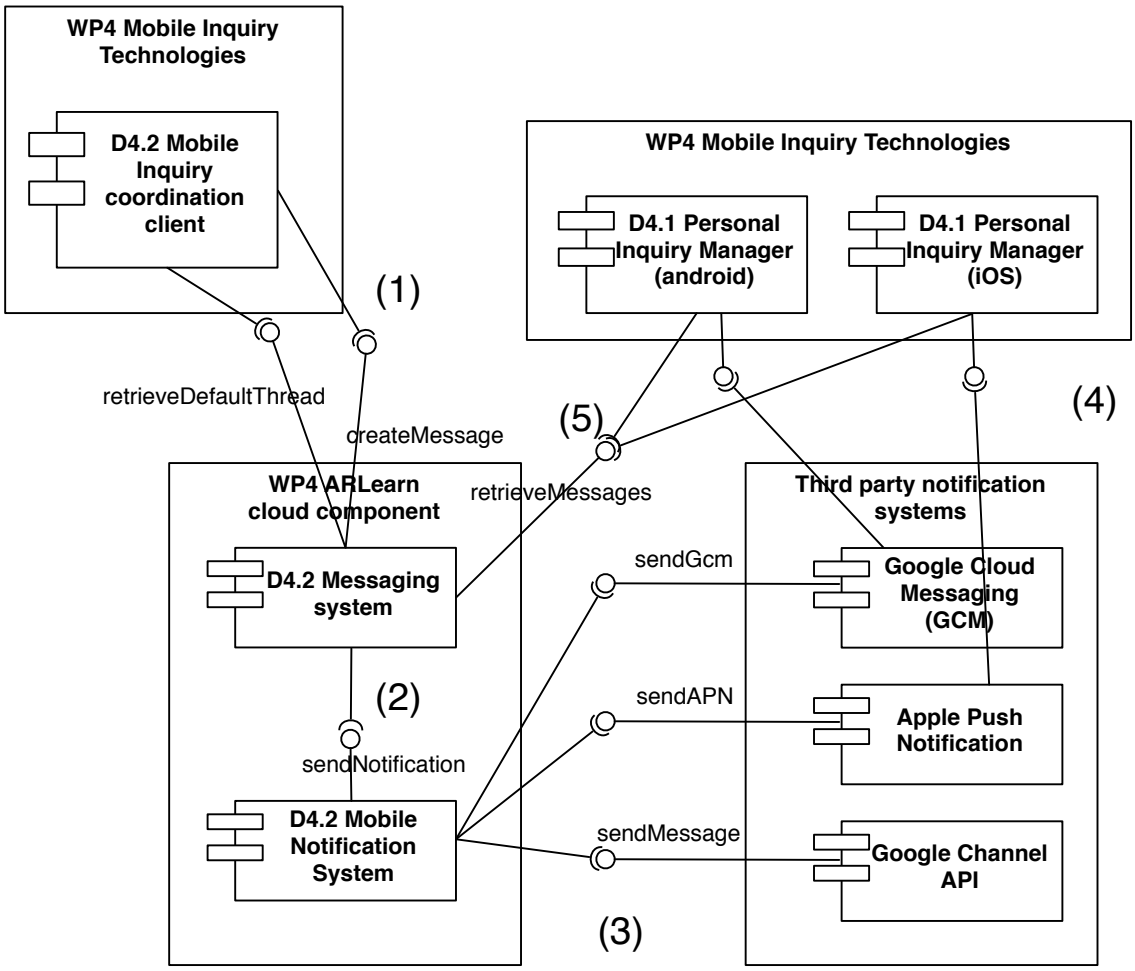

Figure 5: The mobile notification system applied to the Mobile Inquiry Coordination Client

The architecture (Figure 5) depicts user story #41 (US#41, 2013) "As a teacher, I must be able to dispatch a message to all users/ to a group of users/ to a single user". The mobile inquiry coordination client first

retrieves the runId that corresponds to the IWE inquiry. With this runId, this component can retrieve the default thread from the ARLearn messaging system (1). Alternatively, the messaging system enables creating new communication threads. Next the user can post a message through the createMessage service and broadcast this message to a single user / a group of users / all users of the inquiry. Currently only broadcasting to a single user or all users is supported. ARLearn supports sending messages to teams of users. This functionality is however not (yet) integrated in the weSPOT components.

As the messaging system receives this message, it uses the notification system (sendNotification), to inform the PIM that a new message is available for the user. The sendNotification message (2) indicates to the Mobile Notification System, what the scope of the message is (i.e. single user must be notified, a group of users must be notified or all users in the inquiry are to be notified). Next the mobile notification system component retrieves all devices that must be notified and sends them a notification using the corresponding technology (GCM, APN or the channel API) (3). As a result the PIM receives a "send-to-sync" message and updates the message list via the retrieveMessages API that is implemented in the Messaging system (chapter 5).

3.3. The IWE chat widget

Users that are participating in the inquiry via the Inquiry Workflow Engine (IWE) will also be able to receive inquiry related messages. The architecture in this case is a variation on Figure 5.

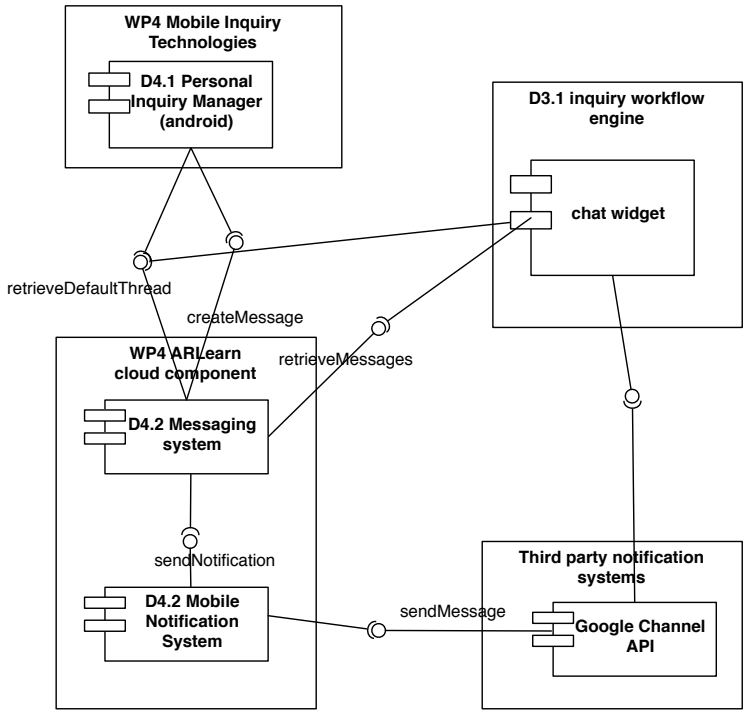

Figure 6: The notification system applied to the chat widget.

Figure 6 exemplifies how a message that is create by the PIM (the android client in this example) finds its way to the chat widget, a web based component. The example therefore does not build on GCM or APN as a means to deliver the notification, but rather utilizes the Google Channel API, because the chat widget is a web-based component.

The chat widget receives notifications from the Google Channel API when a new message is available. The channel API is implemented as a JavaScript library that will indicate the availability of a new message.

Note that in this example the role of the PIM (sender) and the role of the chat widget (receiver) is interchangeable. However, when the chat widget posts new messages, the PIM will be notified via GCM or APN and other users that are subscribed to the thread via the chat widget will receive updates via the Channel API.

4. Conclusion and next steps

This report presents the work conducted so far under Task 4.2 of WP4. At the time writing this deliverable the following implementations have been finished:

- The context-aware notification system as whole, including the framework for sending notifications to mobile and web based components. The mechanism that was described to contextualize data collection tasks is also currently operational.
- Support for notifications is integrated in the PIM. Data collection tasks are update on the PIM via the framework described in this deliverable
- Generic functionality for creating and retrieving messages is available in the PIM. The GUI for this functionality is however at this moment not ready.

The following components will be further developed in the next months to come

- The creation of the chat widget. Work on this widget has started and will be further conducted.
- The MICI client (task 4.4) is currently under development and is due in 6 months.
- Integration context into the IWE widget for creating data collection tasks. Although the functionality is implemented on the PIM, the IWE widget does not yet support defining these context parameters.

In addition we expect these further developments to lead to fine-tuning the API with additional methods.

5. References

Mikroyannidis, A. (2013) D3.1 Inquiry Workflow Engine. Available at http://wespot.net/public-deliverables

Ternier, S. (2013) US#41: As a teacher, I must be able to dispatch a message to all users / to a group of users / to a single user. Available at http://trac.wespot.net/ticket/41

Ternier, S., Charleer, S. (2013) D4.1 Personal Inquiry Manager. Available at http://wespot.net/public-deliverables

Ternier, S., Klemke, R., Kalz, M., Van Ulzen, P., & Specht, M. (2012). ARLearn: augmented reality meets augmented virtuality [Special issue]. Journal of Universal Computer Science - Technology for learning across physical and virtual spaces, 18(15), 2143-2164.

6. Appendix I : Notification System API

The notification system features an internal API and a publicly available web based API.

1. The internal API enables the ARLearn cloud components to notify applications directly when new content is available
2. The public API offers a REST style API through which weSPOT components can send notifications.

All source code is available via http://arlearn.googlecode.com/
Further documentation and examples are available via http://portal.ou.nl/en/web/arlearn/development

6.1. Register Android device

Method: /rest/notifications/gcm
Type: POST

Parameters

- (in) deviceDescription (account, device identifier, registration id)

This method enables registering a user's Android device to the system. This method will store the registration that is issued by the Google Cloud Messaging service and bind this to the user's account and a unique identifier for the device.

6.2. Register iOS device

Method: /rest/notifications/apn
Type: POST

Parameters

- (in) deviceDescription(account, device identifier, device token, bundle identifier)

This method enables registering a user's iOS device to the system. This method will store the deviceToken that was issue by Apple's Push Notification service and bind this to the user's account and a unique identifier for the device. Furthermore, as broadcasting to different applications is supported, the bundleIdentifier (that identifies the software) is registered with description.

6.3. List devices for user

Method: /rest/notifications/listDevices
Type: GET

Parameters

- (in) account
- (out) array of device descriptions

This method lists the various devices that a user has registered in the system. This enables an app to send notifications to a particular device instead of sending a notification to all devices.

6.4. Send notification

Method: /rest/notifications/notify
Type: POST

Parameters

- (in) account
- (in) notification

This method will broadcast the notification message to all devices that are registered with the user's account.

7. Appendix II: Messages API

The messages API is a RESTfull API that enables the PIM, IWE and the MCI to exchange messages. The messages API builds on the internal notification system. As a client component will submit a new message, target tools like the PIM or IWE will get a notification indicating the availability of a new message. The messages API features managing threads and messages. A thread aggregates various messages. Within an inquiry, tools can define an arbitrary amount of threads. Threads are however optional, when no thread is created, a default thread is assumed.

All source code is available via http://arlearn.googlecode.com/
Further documentation and examples are available via http://portal.ou.nl/en/web/arlearn/development

7.1. Threads

A thread is defined by the following properties.

- A *threadId* is a server generated property that uniquely identifies the thread
- The *name* property is a character string that names the thread.
- *Default* is a boolean that indicates whether or not this thread is the default thread
- *RunId* identifies the run within which the thread was created. This runId correponds to one inquiry.

7.1.1. Create thread

Method: /rest/messages/thread
Type: POST

Parameters

- (in) thread (name, runId)
- (out) thread (threadId, name, runId)

This method creates a new thread. After successful execution, this method returns the JSON thread object, including a threadId that is generated by the server

7.1.2. Get thread for a run/inquiry

Method: /rest/messages/thread/runId/123
Type: GET

Parameters

- (in) runId
- (out) array of threads

This method retrieves all threads that correspond to the given run.

7.1.3. Get default thread for a run/inquiry

Method: /rest/messages/thread/runId/123/default
Type: GET

Parameters

- (in) runId
- (out) array of threads

This method retrieves the default thread for the given run. If no default thread exist, a default thread is created and returned.

7.2. Messages

A message is defined by the following properties

- A *messageId* is a server generated property that uniquely identifies the message.
- The *threadId* identifies the thread to which the message is attached.
- The *body* is a character string that details the message
- *Date* is the amount of milliseconds, between teh time when this message was submitted to the server and midnight, January 1, 1970 UTC.
- The run identifier within which the message was created is identified by *runId*.
- TeamIds[] enables sending a message to group of users.
- UserIds[] enabels sending a message to a list of users

When both teamIds or userIds are not specified, the message is visible for all users within the inquiry.

7.2.1. Create message

Method: /rest/messages/message
Type: POST

Parameters

- (in) message (body, threadId)
- (out) message (messageId, body, threadId)

This method creates a message. When no thread is specified, the default thread is assumed. As a result of submitting this message, the message will be sent to the user via the MNS.

7.2.2. Retrieve messages for thread

Method: /rest/messages/messages/threadId/123
Type: GET

Parameters

- (in) threadId (e.g. 123)
- (out) array of messages

This method retrieves all messages for a thread.

7.2.3. Retrieve messages for thread

Method: /rest/messages/messages/runId/123/default
Type: GET

Parameters

- (in) runId (e.g. 123)
- (out) array of messages

This method retrieves all messages for the default thread of a run.