IST Amigo Project
Deliverable D4.7

Intelligent User Services
6 - Community CE-HTML based Experience Sharing Service Software Developer's Guide

IST-2004-004182
Public

Project Number	:	IST-004182
Project Title	:	Amigo
Deliverable Type	:	Report

Deliverable Number	:	D4.7 (CHESS contribution)
Title of Deliverable	:	6 - CHESS Developers guide
Nature of Deliverable	:	Public
Internal Document Number	:	amigo_6_d4.7_final
Contractual Delivery Date	:	30 November 2007
Actual Delivery Date	:	14 January 2008
Contributing WPs	:	WP4
Author(s)	:	Peter Vink (editor), Peter Lambooij, Dietwig Lowet, Ingmar Struijs, Maarten van Lijden, Jaap van Otterdijk, Bart Poirters (Philips)

Abstract

This document is the developers guide for the Community (or Content) Sharing Service components, it describes how to install, use and edit the components of CHESS

Keyword list

Ambient intelligence, sharing activities, communities, jabber, XMPP, CE-HTML, software developers guide

Table of Contents

1 Component Overview

The CE-HTML based Experience Sharing Service (CHESS) provides a generic mechanism to share web based applications between multiple clients. The basic principle of this mechanism is to provide community based aspects to these applications, allowing sharing of information and experiences.

The standard that has been chosen for rendering of the User Interfaces is CE-HTML [ref CE-HTML]. This format, which closely resembles HTML but is more specific to CE equipment, can be used by applications to create a GUI that can be displayed by various displays in an Amigo home, the input and output capabilities will then depend on the device (e.g. screen resolution, touch screen, remote control).

The standard that has been chosen for sharing presence and for sending messages between homes is XMPP [ref XMPP] (also known as 'Jabber'), which matches with the solutions chosen in WP7 demonstrators.

Figure 1-1 gives a rough overview of how the Community CE-HTML based Experience Sharing Services operates within the Amigo network.

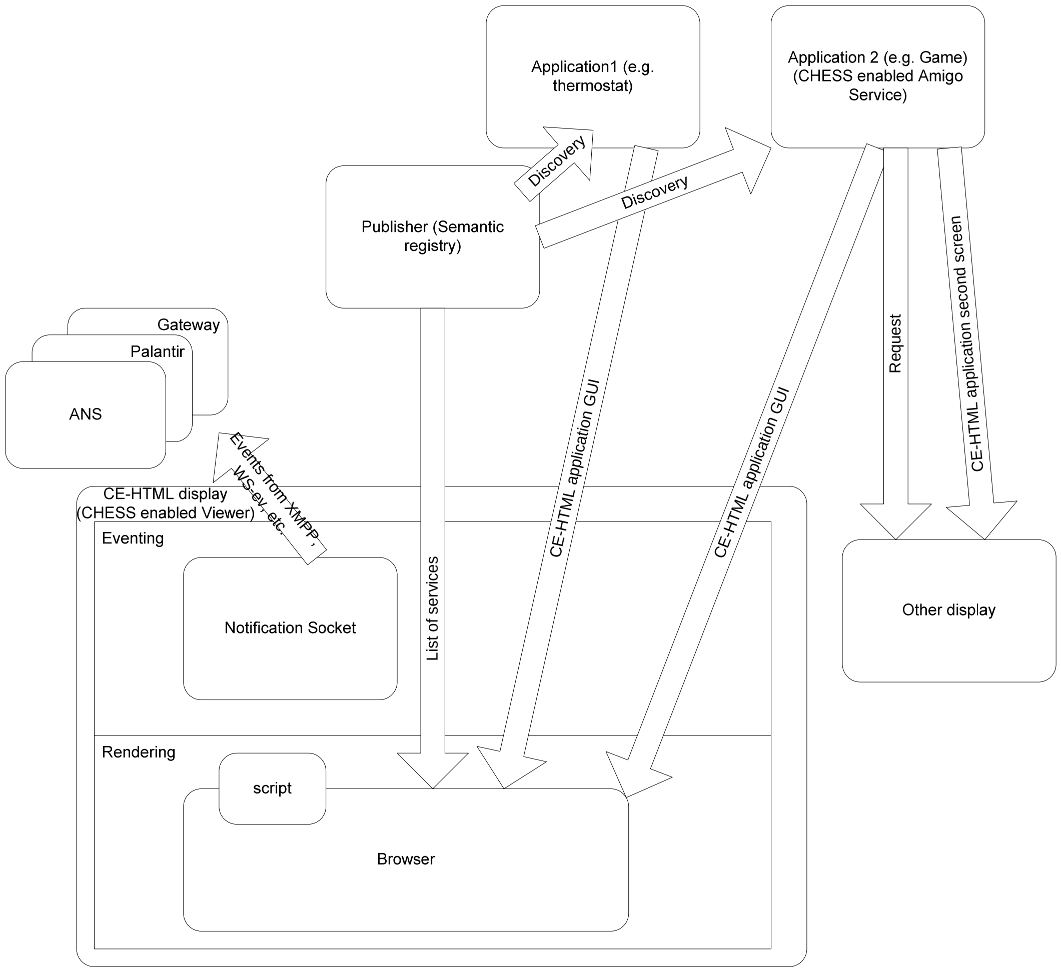
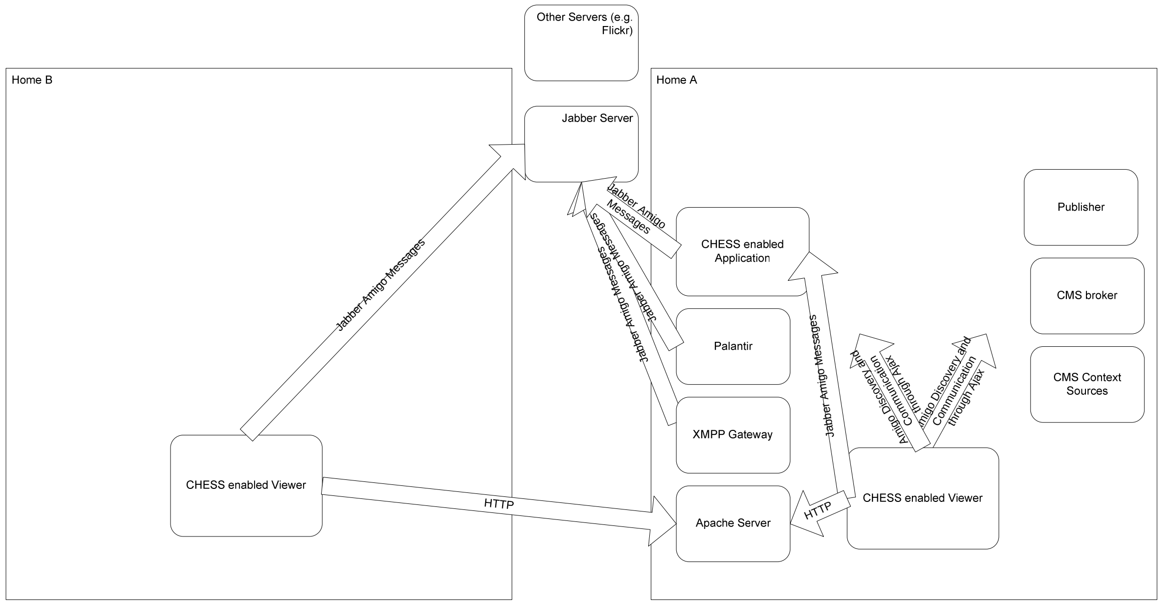

Application1 (e.g. thermostat)
Application 2 (e.g. Game) (CHESS enabled Amigo Service)
Discovery
Discovery
Publisher (Semantic registry)
Gateway
Palantir
ANS
Events from XMPP, WS-ev, etc.
CE-HTML display (CHESS enabled Viewer)
Eventing
Notification Socket
List of services
CE-HTML application GUI
CE-HTML application GUI
Request
CE-HTML application second screen
Other display
Rendering
script
Browser

Figure 1-1 Overview of Sharing Services together with other Amigo Services

In this figure, Applications (such as a game, or a thermostat) export their graphical interfaces as CE-HTML pages that can be accessed in the Amigo network. These applications can indicate that they export a CE-html interface using Amigo semantic discovery and/or context information. This data is stored a combined in a publisher service (which acts as the registry). Another CE-HTML based application, running for example on a large screen can then find these services and link directly to their pages (using same mechanism).

In addition, modules are provided to access the Publisher Service and other Amigo services from a CE-HTML browser and to display pages on remote displays (provided that the service running on these screens will accept these pages). An example will be provided that runs on a main screen but uses a second screen to provide additional information.

The next Figure shows how the services are used in the extended home scenario.



Since both the viewer and the Service have been extended with XMPP functionality they can also choose to communicate to each other through XMPP. In the example above a service is hosted in Home A . This service can be accessed through its CE-HTML interface which is published on a (regular) Web server. Furthermore, the service can use XMPP to locate other Users, Devices and Amigo Services. The minimal set required in a second home (Home B) is shown above, this is the scenario that has been implemented in the example client discussed below. Of course, Home B can also run its own Amigo services.

Community Sharing Service consists of the following components; though these components can be regarded as services by themselves, the CHESS is provided as one single service. The example applications shows how the separate parts work together.

Rendering software / Scripts Typically this software consists of a standard browser with standard plugins. Javascript components are provided for communication to components such as the notification socket and other Amigo services (CMS, ANS, …)

NotifSocket This component provides an eventing mechanism to the GUI renderer. Its functionality can be further divided:

Event synchronization. This component handles synchronization of events, mostly these are UI events.

Multi channel Administration. Handles administration for setting up several channels between users.

Community Handler (e.g. via XMPP) Publisher This component creates the link between applications providing CE-HTML and applications rendering CE-HTML

Publisher(Service Registry) Maintains a list of all published services / applications, thus enabling 'amigo' discovery for browser based applications. Uses (simplified version of) the Amigo Semantic Service descriptions.

Gateway Enables communication between homes.

Example application(s)/ Client. Example applications will fit into the Shared Activities scenario of WP7. The example client contains re-usable components for communicating with User Applications (For example sending events to update specified screen elements) Example client will consist of two parts: A part running on the device (written in javascript) and a part running 'somewhere' in the network, the second part has been implemented as an Amigo service.

These services were designed specifically to support devices with GUI based on CE-HTML and using XMPP for communication protocol.

1.1 NotifSocket

Provider

Philips

Introduction

The NotifSocket is a standard component of a CE-HTML compliant browser [CE-HTML]. It is offered by the browser as a java-scriptable object. Through javascripting a persistent tcp/ip connection between the client and a server can be set-up. This allows for an arbitrary asynchronous transfer of information between client and server. An important use case is server push of information to the client. The NotifSocket is unique in this respect. No other browser mechanism (including XMLHTTPRequest) allows for interleaved asyncronous messages between server and client. In the present use case NotifSocket is utilised to establish a permanent tcp/ip connection to presence/messaging services e.g. XMPP [XMPP]

The package delivered contains an installable plugin together with it's source code. An example application (test_notifsocket.zip) is provided to show how the combination of CE-HTML, javascript and notifSocket can provide a resource binding between an (in-home) client and an XMPP derver. A document of how to install and operate the example application is included in the zip file.

Development status

Version 1.4 available (as a plugin for Mozilla Firefox)

Intended audience

Developers

License

Philips License, BSD style

Uses Gecko-sdk, licensed under MPL (Mozilla Public License)

Language

C++

Environment (set-up) info needed if you want to run this sw (service)

IBM compatible PC, Mozilla browser 1.5 - 2.0.

Platform

The software has been tested on a PC running under Windows XP SP2.

Tools

Microsoft Visual Studio

Files

reference to zip or url

Documents

Designers guide is this document.

Tasks

?

Bugs

There is one known bug. Javascript callbacks from NotifSocket to the browser to indicate arrival of data on the socket sometimes cause the browser to crash. A work-around for this problem can be found in the test page test_notifsocket.html.

Patches

None

1.2 Publisher

Provider

Philips

Introduction

Maintains a list of all published services / applications, thus enabling ‘amigo’ discovery for browser based applications. Uses (simplified version of) the Amigo Semantic Service descriptions. Exports the sparql based Amigo Context interface and a second, more specific, interface to filter and list available services in a simple way.

Development status

Available

Intended audience

Developers

License

Philips License, BSD style

Language

C#

Environment (set-up) info

PC, Amigo .net framework

Platform

Windows

Files

Explained in section 4.3.3

Documents

This document

Bugs

None yet

Patches

None yet

1.3 Example Clients (poker game & Quiz game)

Provider

Philips

Introduction

The example client shows two possible implementations of an extended home application (ie a poker game and a quiz). The application uses a generic XMPP based framework to exchange messages between users and devices in different homes. In the poker game, html screens are generated as interface to the users. These screens can be shared, which means that several users are watching the same screen and browsers are kept in synch via dedicated messages. The backend of the example client is written as an amigo service and able to use various other Amigo services.

Development status

Available

Intended audience

Developers, poker players ☺

Licence

Philips Licence, BSD style

Language

Javascript (browser side) and C# (Amigo service side)

Environment (set-up) info

Javascript part was tested for Mozilla Firefox using Firebug. Client needs NotifSocket.

Apache server.

Jabber Server.

Platform Hardware

Interface: Any platform with browser and xmpp plugin (e.g. PC or PDA)

(Amigo) service: runs on windows

Documents

This document

Bugs

No bugs reported for now

Patches

No patches created for now

1.4 XMPPGateway

Provider

Philips

Introduction

A gateway that can accept an xmpp message for discovery or invoking of a local amigo service or that passes such message on to a remote home.

Development status

First version available

Intended audience

Developers

Licence

Philips Licence, BSD style

Language

Java

Environment (set-up) info

Runs in OSGI container (OSCAR), requires JVM

Needs jabber server

Platform Hardware

PC

Files Source code

See [ius]/sharing service/XMPPGatewayService

Documents

This document

Bugs

No bugs reported for now, note that this service is not ‘secure’

Patches

No patches created for now

2 Deployment

An operational service typically consists of a number of renderers (browsers), a publisher and a number of services that provide a CE-HTML based GUI (in this case the poker game, but more are possible). Below is shown how to run each component

2.1 Setting up the Community based Experience Sharing Environment

In this section we present, as an example, the environment in which the service has been tested (PC platform). Reader should realize that other solutions exist.

(copy the install stuff)

- Mozilla – Used as browser

  (www.mozilla.org), install directly from mozilla: version used for testing 1.5.0.9 and 2.0.0.6

- VLC Used to emulate tv image and to display video connections

  http://www.videolan.org

- Autohide – used to hide toolbars

  (http://www.krickelkrackel.de/autohide/) install by running autohide.xpi

- keyconfig – used to disable predefined keyboard shortcuts

  http://extensionroom.mozdev.org/more-info/keyconfig

Typically, the base of the application exists only of the software above and notifsocket, together with hardware for receiving Infra Red and translating this into events. The rest of the application is provided as Ce-HTML pages / javascript.

It is possible to use an infrared receiver and remote control, though demo operates with keyboard.

2.2 NotifSocket

2.2.1 System requirements

Any system that can run a browser (currently limited to Firefox)

(external) jabber server

2.2.2 Download

Notifsocket.xsi

2.2.3 Install

Install as a mozilla plugin, alternatively, you can take the next steps

- Copy the . xpt-files to the folder ‘components’ that you find in the folder Mozilla Firefox
- Copy the .dll-files to the folder ‘plugins’ that you also find in the folder Mozilla Firefox.
- Restart Firefox.

2.2.4 Configure

-

2.2.5 Compile

Compiles under visual studio

2.3 Publisher

The publisher component is not an integral part of the sharing service. However, it is a necessary component when a User wants to use the example application (see below). Therefore, the publisher component has been made available. Note that at the moment of its release, this service does not (yet) use the actual semantic registry (see Ami05f). It is not advised to keep two independent regisitries.

2.3.1 System requirements

Windows / Amigo .net framework

2.3.2 Download

See [repository]/ius/community sharing/DeviceServiceRegistry/

2.3.3 Install

Runs as executable

2.3.4 Configure

Needs no configuration, but needs some services to register. A small tool was created to facilitate this (see appendix)

2.3.5 Compile

Written in Visual studio, comes with .sln file

2.3.6 Open tasks

Does not yet use semantic registry (as defined in D3.4)

2.4 Example Service (PokerGame)

2.4.1 System requirements

GUI: Any platform that can run browser with NotifSocket (see above)

Amigo Service: PC/Windows

(external) Jabber Server

(external) Web Server (eg Apache)

2.4.2 Download

Software contains two parts:

Gameservice is an amigo service, using the C# framework (also provided through dlls)

HTML files (these are found in the html directory in the two subdirectories ‘pokerlobby’ and ‘pokergame’

2.4.3 Install

Amigo service can be installed from the .msi file

HTML files need to be published on a web server such as apache

Game Service will try to discover ADP (registry) and Context Broker (Context Management) through the broker it will look for location information of persons and (portable) devices

2.4.4 Configure

Game service can be configured through the file ‘jabbersettings.xml’ in the root directory

In this file, the following parameters can be given

CommunityHost	(Extended) Jabber host IP address (not used in this set up)
Host	Jabber host IP address
JID	JID that gateway will use
Password	Password that gateway will use
WebHost	Web Host IP address (assumes that HTML part of the application is present on this host)

2.4.5 Compile

GameService has been developed using Visual Studio (.sln file is provided)

2.5 XMPPGateway

2.5.1 System requirements

Any system running OSGI container (eg pc)

(external) jabber server

2.5.2 Download

Implemented as an OSGI Amigo bundle

Requires only one jar: XMPPGatewayService.jar

2.5.3 Install

Install through OSGI container (eg OSCAR)

2.5.4 Configure

XMPP gateway can be configured through the file ‘jabbersettings.xml’ in the root directory

In this file, the following parameters can be given

CommunityHost	(Extended) Jabber host IP address (not used in this set up)
Host	Jabber host IP address

JID	JID that gateway will use
Password	Password that gateway will use

2.5.5 Compile

Written in java using Eclipse (eclipse project file is also provided)

Uses Smack libraries, written by jive software

2.6 Dependencies

The shared services have the following dependencies:

Jabber Server is needed for communication of presence and for exchanging the messages specified in this document.

HTTP server is needed for hosting the CE-HTML pages.

3 Component Architecture

3.1 Interfaces

Notifsocket

The Notifsocket can not be accessed as an Amigo Service. Its functionality can be used in javascript to access other services and receive events back. Exported functions are:

```
public Bool   init_socket(SOCKET client);
public Bool   connect_socket(SOCKET *client, char* IPaddr, short port);
public int    poll_socket(SOCKET client, char* buffer );
public int    send_socket(SOCKET client, char* msg );
public int    close_socket(SOCKET client );
```

Note that in the example application two javascript files are provided (jabber.js and jabberhandler.js) that add functionality to the notifsocket (such as, for example, reconstructing an xmpp stanza from incoming data), These components are presented in the next chapter.

Publisher

Publisher service is intended as a simple registry for OWL-S-Amigo descriptions of a service

```
public String Register(String serviceDescription);
public String Deregister(String serviceDescription);

public String setStatus(String deviceName, String deviceStatus);
public String setUsedBy(String deviceName, String deviceUser);
public String setInterrupt(String deviceName, String deviceInterrupt);


public String getStatus(String deviceName, String deviceStatus);
public String getUsedBy(String deviceName, String deviceUser);
public String getInterrupt(String deviceName, String deviceInterrupt);
public String getServices(String serviceDescription);
public String dumpAll();
```

The first two functions are already sufficient for using the registry, other functions were added to facilitate manipulation of a service's status without having to resend the entire ontology.

Publisher is also a context source, which means that it also exports the standard Amigo iContextSource interface:

```
public String query(String contextQueryExpression);
public String subscribe(String contextSubscriptionCharacterisation,
      String contextSubscriptionReference);
public boolean unsubscribe(String contextSubscriptionID);
```

The sparql based query (and subscribe) functions offers a complete interface for getting all service data, the get functions above offer a direct simple but limited interface for applications that prefer this.

Gateway

Gateway has only one function, this function accepts messages that are defined as *AmigoRequests* see below.

```
public String callRemote(String amigoAction);
```

name space is *urn:amigo,* service type is *XMPPGatewayService*

Next to their service interfaces, several of these services also accept messages sent over XMPP (to their respective JID defined address) these messages have been formatted as *AmigoActions,* which is explained below.

Example Client

Example Client can be discovered as an Amigo Service, Most of its functionality is accessed through XMPP messages (see below), it exports only one interface through SOAP:

```
public String getUserLocations();
```

which can be used to get insight in the status of the poker game.

XMPP interface:

Gateway and Example Client also accept XMPP messages as defined below.

3.1.1 CE-HTML

The CEA-2014 Web4CE standard allows a UPnP device or an Internet Service (acting as a CE-HTML web server) to transport a rich graphics-based interactive user interface in the form of CEA-2014 compliant HTML pages. Web4CE gives UPnP device vendors as well as authors of Internet Services full control not only over what functions can be remotely controlled but also over the look and feel of the user interface that will be shown to the consumer.

In the Amigo context, CE-HTML has been enabled by running a Web server inside the home, Amigo Services that export an HTML inter based interface can use this server. A specific capability was added for this to the Amigo service ontology (CE-HTML interface, which refers to a url within the home with possibility to access the server from other homes)

3.1.2 AmigoRequests (XMPP)

The components that have an XMPP interface (Gateway, ExampleClient) accept a number of specific messages, these are described in this section.

The Gateway component accepts XML packets that are defined in the following way:

```
<?xml version="1.0" encoding="utf-8"?>
<AmigoRequest>
     <From>peterv@130.145.232.74</From>
     <To>petervsgateway@130.145.232.74</To>
     <Service>Framework</Service>
     <Function>Discover</Function>
     <Param0 name="namespace" >urn:amigo</Param0>
     <Param1 name="servicename" >ContextBroker</Param1>
     <Param2 name="timeout">3</Param2>
</AmigoRequest>
```

with the following meaning

From	JID of sender (or any identifier when callremote is used)
To	JID of the remote home
Service	Service name (name local to remote home)
Function	Function to invoke (at Service)
Param1	first parameter of function to invoke
Param2	Second parameter of function to invoke
	and so on…

The functionality provided by the Amigo framework is also supported by setting Service to ‘Framework’

Allowed functions to call are ‘Discover’ (=lookup function of LDAPlookup) , ‘DiscoverFirst’, ‘Subscribe’, ‘Unsubscribe’ and ‘Notify’

These packets are accepted as input for the callRemote (amigo/soap) interface and on the jabber interface

The JID of the gateway can be configured by changing the file jabbersettings.xml

The example demonstrator contains a part that is running inside a browser and an (Amigo) web service. Both parts accept specific xmpp messages. These messages also constitute the base of how the two parts work together (next to the AJAX interface).

Messages that are accepted by the javascript part:

```
<AmigoRequest>
   <From>peter@amigo</From>
   <To>Jerry@amigo</To>
   <MessageType>showMessage</MessageType>
   <Param0>general message</Param0>
   <Param1>messageHTML</Param1>
</AmigoRequest>
```

where messageHTML is an html form

```
<AmigoRequest>
   <From></From>
   <To></To>
   <MessageType>redirectPage</MessageType>
   <Param0 name="pageName">Poker game</Param0>
   <Param1 name="pageLink">
      http://130.145.232.193/PokerGame/index.html
   </Param1>
</AmigoRequest>
```

```
<AmigoRequest>
   <From>xmppUser@xmppServerName/xmppResource</From>
   <To>deviceName@xmppServerName/xmppResource</To>
   <MessageType>updatePage</MessageType>
   <Param0>divName</Param0>
   <Param1>divContent</Param1>
   <Param2>divType</Param2>
</AmigoRequest>
```

some messages also have specific replies.

The table below shows the messages that are accepted in the example application

Component	**Accepted Message**	**Comment**
PokerGame (Amigo Service)	Start	Start the game from GUI
PokerGame (Amigo Service)	Stop	Stops and resets the game, for debug reasons
PokerGame (Amigo Service)	*Other game specific message*	*These messages are: login user, logout user, accept inivte (conditional login) handle useraction (fold,bet,raise) getAllPlayers, getUserStatistics*
PokerGame (viewer)	Redirect	Switch between Menu, lobby, game
TVMenu (viewer)	Redirect	Switch to external application (eg poker game in other home)
TVMenu (viewer)	Message	Show invitation message
PokerGame (viewer)	Message	Show game related messages
PokerGame (viewer)	updatePage	Show game related updates
PokerGame (viewer)	dataRequest	Request for data from Viewer side (passed to current application)
PokerGame (Amigo Service)	dataRequestAnswer	Answer to the above
Photoframe	Redirect	Switch to external screen (eg poker game)

3.1.3 Service Registration

The publisher accepts input of type *ServiceDescription* for registration, deregistration and status updates of services. The format is based on the Amigo Service ontology as explained in D3.2. Below is an example of how the ontology was sued in the example.

```
<rdf:RDF
  "xmlns:amigo="http://amigo.gforce.inria.fr/ontologies/amigo#"
  "xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  "xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  "xmlns:owl="http://www.w3.org/2002/07/owl#"
  "xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  "xml:base="http://amigo.gforce.inria.fr/ontologies/amigo#">

  <Service rdf:ID="Service">
    <Name>Poker game</Name>
    <Address>http://HostAddresses:Port/Pathname</Address>
    <Status>Available</Status>
    <Interrupt>TRUE</Interrupt>
    <UsedBy>None</UsedBy>
    <hasCapability>
      <Capability>
        <hasInput>
           <Input>
              <Type>XMPP</Type>
              <Interruptable>gameservice</Interruptable>
              <!—mention which messages are accpted →
           </Input>
        </hasInput>
        <!—can also add the key input here (ie accepts remote) →
        <hasOutput>
           <Output>
              <Type>CE-HTML</Type>
              <Page>http://hostname/PokerLobby/index.html</Page>
              <!—More info on the HTML page here eg size →
           </Output>
         </hasOutput>
       </Capability>
     </hasCapability>
  </Service>
</rdf:RDF>
```

From this example it can be concluded that only a very small subset of the ontology is used, no preconditions and post conditions have been defined and the protocols are mentioned as data rather than connectors. However, the scheme is extensible and the choice for the Amigo vocabulary keeps these possible extensions open, including use by reasoning engines.

3.2 Mechanisms of interaction

Figure 3-1shows how the Amigo messages can be used for a shared activity. Note that other scenarios are also possible, this figure is intended as an example. In this example there is a shared Amigo Service (a game) that is installed somewhere in an Amigo home A. This service will register itself at the semantic registry in the Amigo home (in this example the semantic registry is accessed through the Device registry Service, introduced in paragraph 1.2) and to a jabber server. After this, user Jerry who lives in home A switches on a shared device/interface (e.g. a television). The (interface application on) television searches the registry and finds the newly installed service. Furthermore, The application also checks who is currently using it, through CMS context sources.

When Jerry now decides to start this (shared) service, the service presents a list of all his contacts, filtered for those contacts that share interest for the current Service (game). This (sub-application) has been co-developed with the Amigo extended home scenario (awareness globe) and is specified in more detail in the corresponding documentation. This list is also able to show information about the contacts such as their availability (location, activity). The service communicates with the interface via the Amigo messages. First, the Service asks the interface to go to the selection screen (via redirect message) and then the interface asks the service for a list of people that share the same interest (via datarequest message).

Jerry can now send an invitation to John, who is in another home and also has an interface running (note that if this was not the case, Jerry would be able to notice this provided that John has allowed this information to pass). The invitation to join is handled through another Amigo message (showmessage) and the reply is treated as another message (answer). After Jerry and John have both joined, their status (activity) is updated by the shared service.

When enough players have joined and the game is actually started (by Jerry) the game will check at CMS level (and possibly combined with the service registry) to see if there are other devices available in the homes A and B that can be used for additional game effects, the example uses a second screen and a colored light. Access to Home B is done through the gateway. In this example, John has only a simple Amigo network (only the interface and CMS) and no devices are found even though Johns UMPS profile allows use of such devices. The game shows him a message explaining that no devices were found through the ShowMessage message.

In the home of Jerry both devices (2nd display and light) are present and can be used (not busy), other scenarios are also possible.

The scenario can continue but the example ends here. Not shown in the figure is that the shared service uses the UpdatePage Message in order to show game progress on both interfaces. Furthermore, the interfaces use http in order to retrieve screens from the service (also not shown).

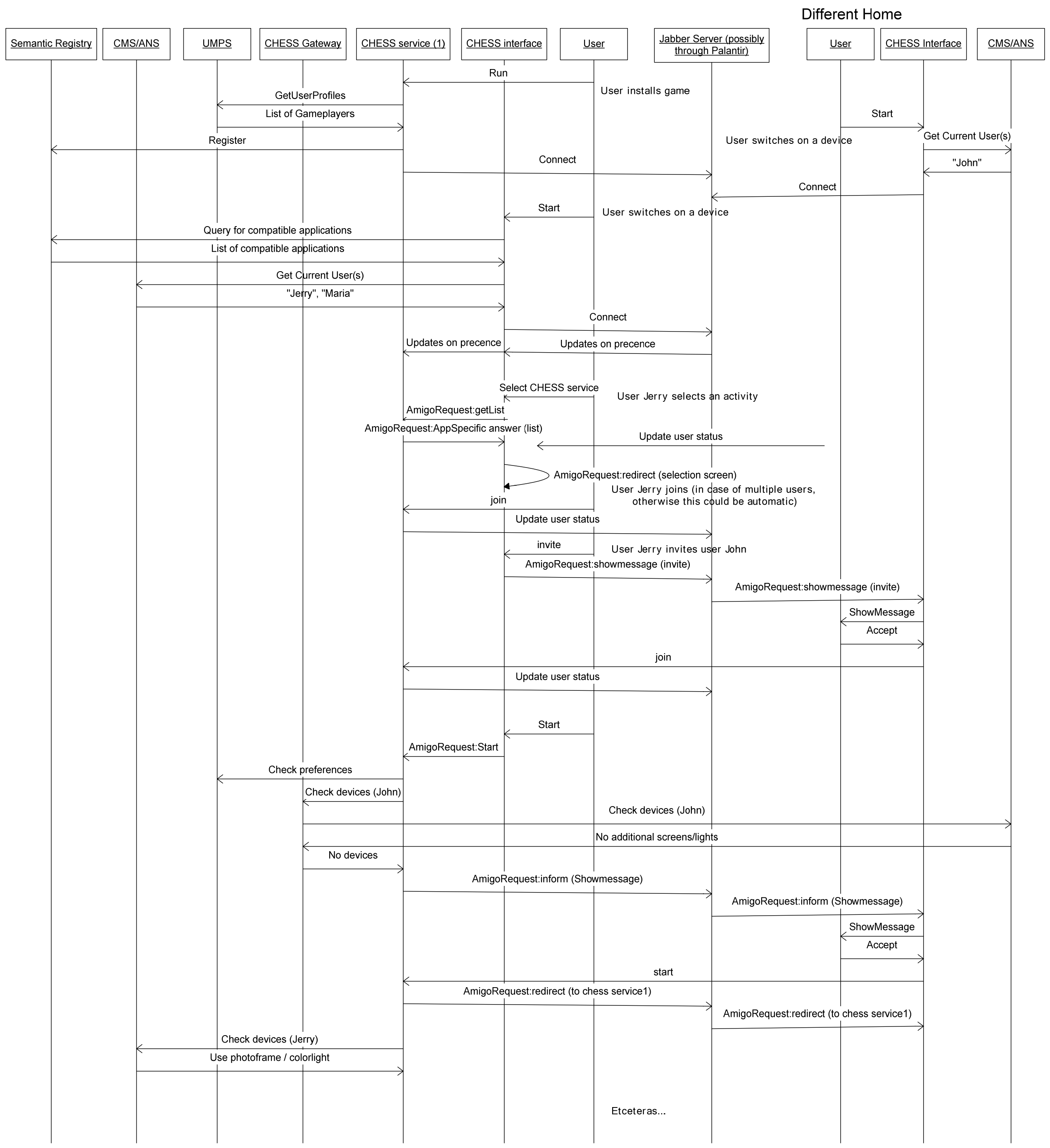

Different Home
Semantic Registry
CMS/ANS
UMPS
CHESS Gateway
CHESS service (1)
CHESS interface
User
Jabber Server (possibly through Palantir)
User
CHESS Interface
CMS/ANS
Run
User installs game
GetUserProfiles
List of Gameplayers
Start
Register
User switches on a device
Get Current User(s)
Connect
"John"
Connect
Start
User switches on a device
Query for compatible applications
List of compatible applications
Get Current User(s)
"Jerry", "Maria"
Connect
Updates on precence
Updates on precence
Select CHESS service
User Jerry selects an activity
AmigoRequest:getList
AmigoRequest:AppSpecific answer (list)
Update user status
AmigoRequest:redirect (selection screen)
User Jerry joins (in case of multiple users, otherwise this could be automatic)
join
Update user status
invite
User Jerry invites user John
AmigoRequest:showmessage (invite)
AmigoRequest:showmessage (invite)
ShowMessage
Accept
join
Update user status
Start
AmigoRequest:Start
Check preferences
Check devices (John)
Check devices (John)
No additional screens/lights
No devices
AmigoRequest:inform (Showmessage)
AmigoRequest:inform (Showmessage)
ShowMessage
Accept
start
AmigoRequest:redirect (to chess service1)
AmigoRequest:redirect (to chess service1)
Check devices (Jerry)
Use photoframe / colorlight
Etceteras...

Figure 3-1Possible interaction between homes using CHESS services

3.3 Writing a “sharing” interface – html side.

On HTML side (shared interface using HTML), the browser will at least have the following components, typically defined in separate frames:

- Communication Frame, which provides the connectivity (ie login) for one or more users

- Application Frame
- Main frame, which provides a functionality from a selected number of Amigo Services as Ajax calls (can be changed extended etc), it contains the messagebox and provides the glue between the two frames mentioned above

It is up to the application to define what else is in the Main frame, typically it could contain a 'main menu'.

The applications that are loaded into the application frame can use the following functionality from the communication frame

GetLoggedOnUsers

SendMessage

….

On the other side, when applications want to support the full set of Amigo messages they should have the following functions available:

- HandleReply
- (UpdatePage) The update Message will directly assume that elements exist in the app frame, depending of the element type, this element is :
    - 1: Text (InnerHTML is updated)
    - 2. Image (Src attribute is updated)
    - 3 Any variable (direct assignment, app should have a function handleAssignment)

4 Example application

The CHESS libraries and services constitute only a small part in the entire Amigo middleware stack, there is no tutorial available but, on the other hand, there is an example application. This application is made up of two parts: An Amigo (Web) Service and a GUI build in Javascript. Both parts use XMPP libraries for communication between themselves and between extended homes.

In this chapter, we present the design of the example application that was introduced in Figure 1-1, Using the explanations given here and the code provided, Readers can construct their own Shared Services.

4.1 System overview

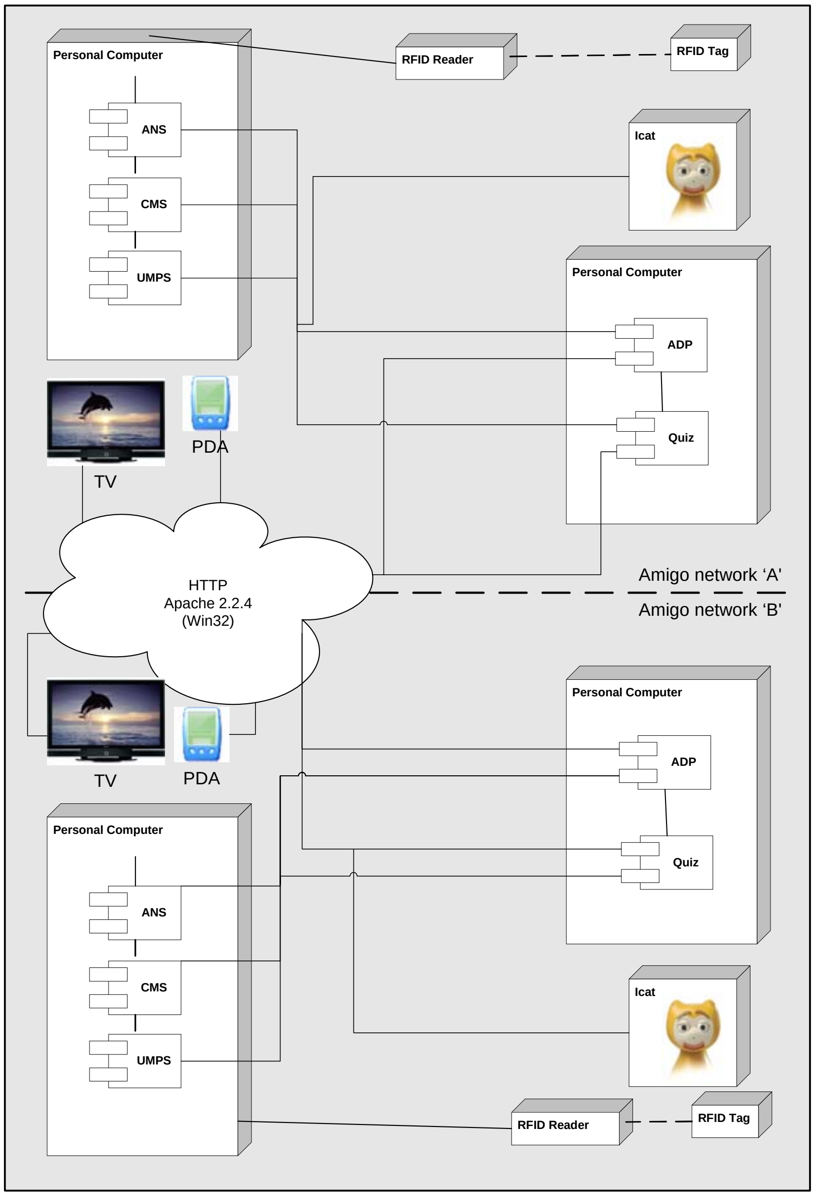


Figure 4-1 Deployment diagram / Component Diagram

In *Figure 4-1 Deployment diagram / Component Diagram* is an overview shown of the components that are related to this design. The design describes how all components talk to another and which protocol is used.

The top of the diagram is an Amigo network and the bottom of the diagram shows an Amigo network. The components in the middle are meant for communication between these networks.

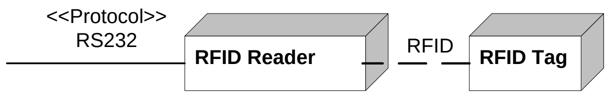


Figure 4-2 Component RFID Components

There is at least one pc in the Amigo network with a location service installed. This service finds all RFID-tags in the neighbourhood by an RFID-reader connected to the pc with a RS232 connection.

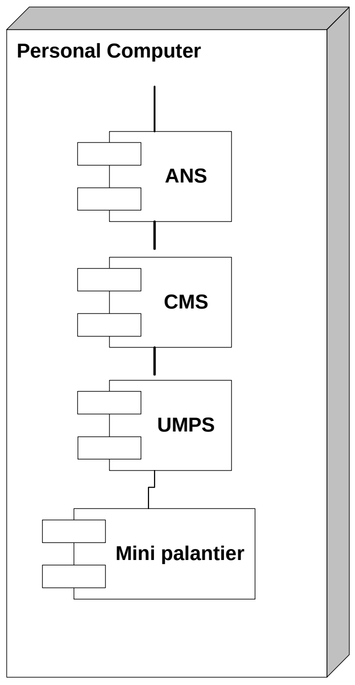


Figure 4-3 Component Amigo PC with services

The Personal Computer is a pre-installed Amigo pc where several services already running. The services [ANS], [CMS] [UMPS] and [XMPPPresence] are the main used services by the ADP and Game Service.

[ANS] Awareness Notification Service
This service uses the information or context from the Amigo network to inform other parties of pre-defined events. (E.g. Jan enters living room)

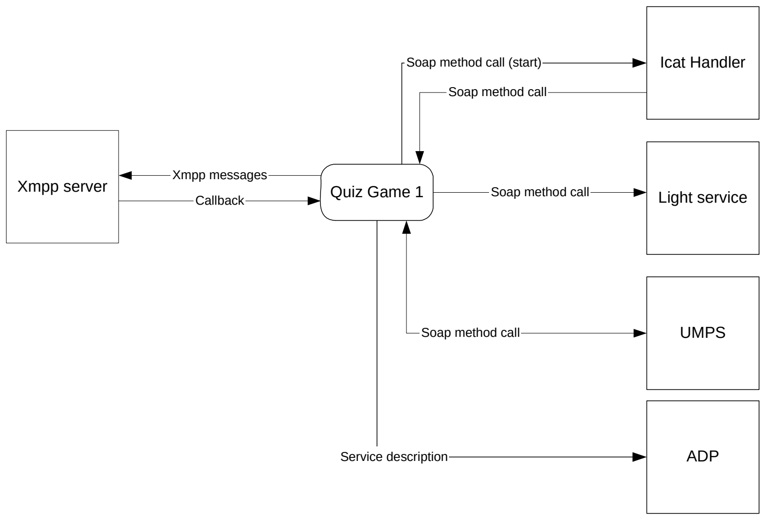
[CMS] Context Management Service
This service collects all context source information and kept it to be queried by other services.

[UMPS] User modelling and Profiling Service
This service keeps profiles from all Amigo network members. It collects preferences from all members but most important it contains the Quiz statistics from all players.

[XMPP precense]
This service collects all information about users. At the moment this is the location, activity and status. The first will be get from the context source. The other two are read from the [xmpp] server all users login to. To get the information this service has simply log in to the xmpp server. Every time there is a change the information will be broadcasted.

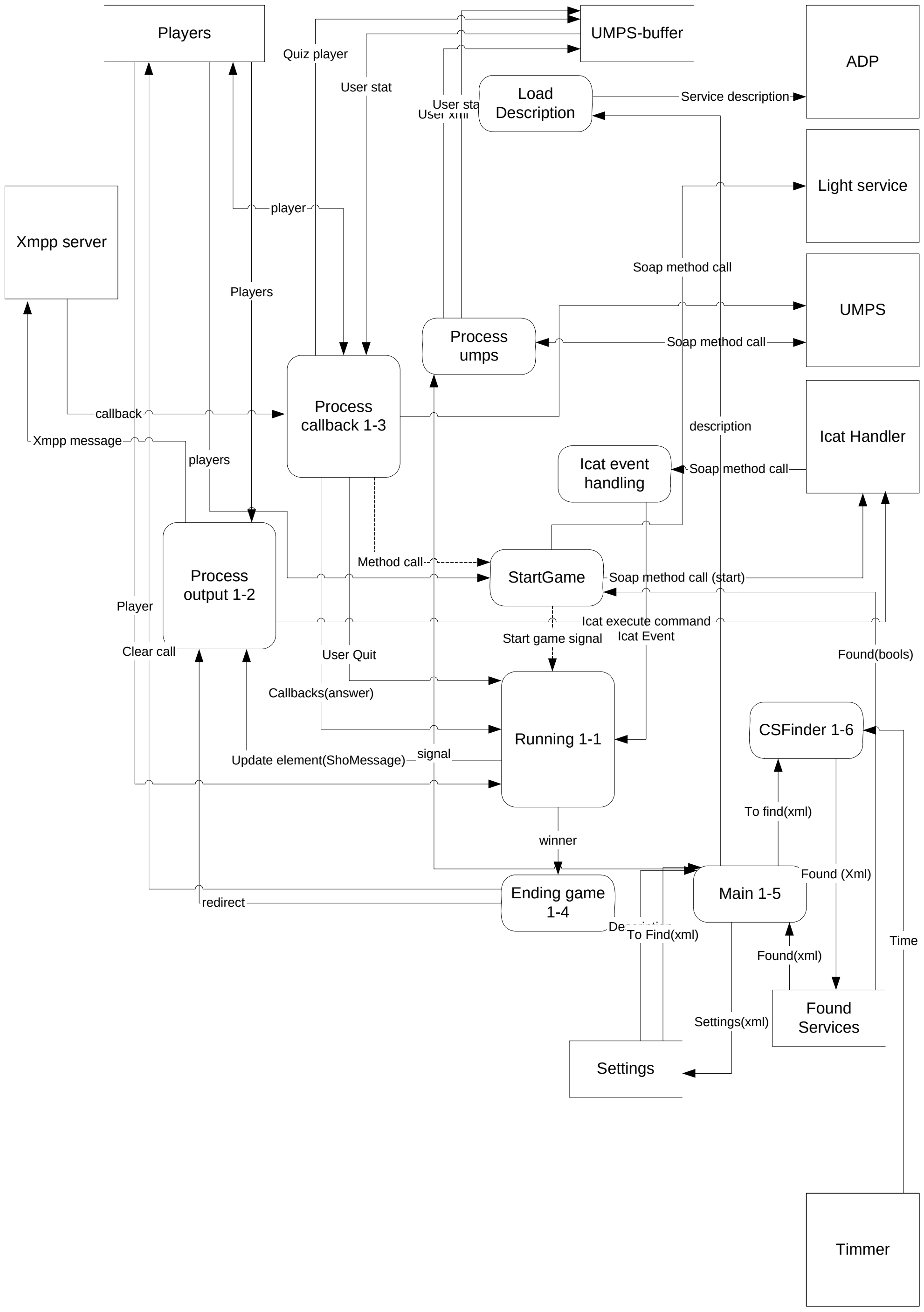
4.2 Data flow diagrams

The following diagram gives an overview of the system with all terminators that are used. The quizgame is our central process. This process contains several sub processes. The most important processes and data flows will be described in the diagrams this chapter contains. To try the diagrams readable the button clicks of the users are grouped as a Call-back message.



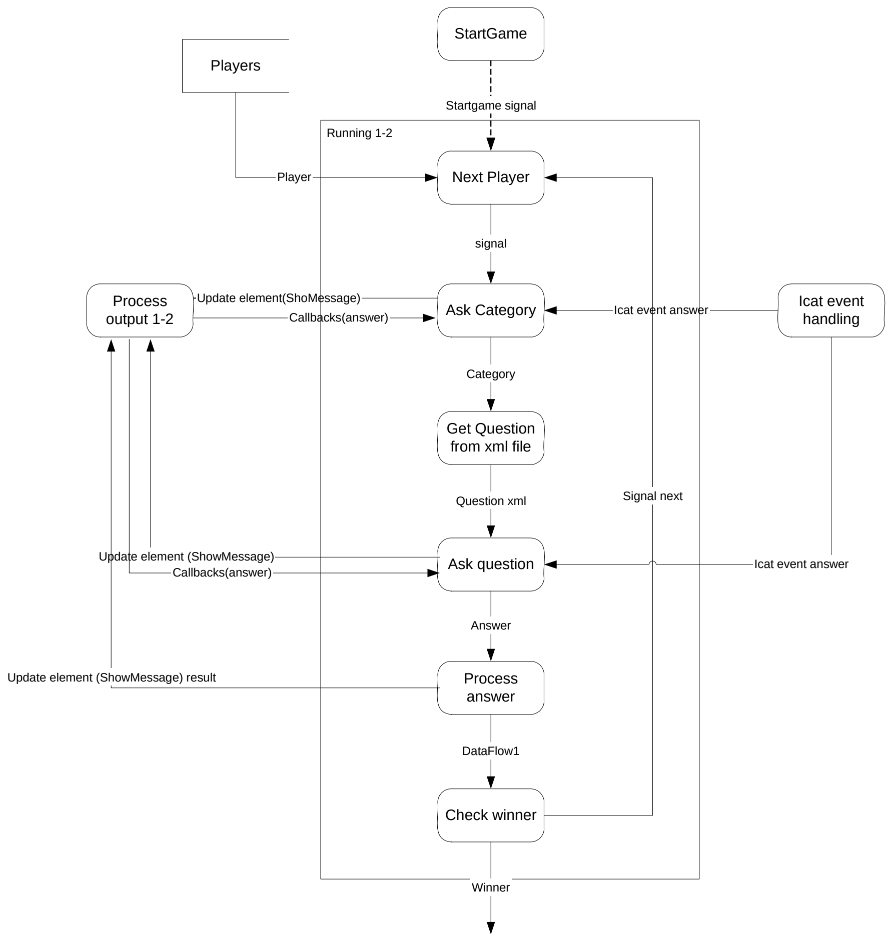
4.2.1 Quizgame 1

This diagram shows us the most important processes of the quiz. Later some sub processes will be described. The Process call-back process is the gate to the outside world of the game. This process is the front door of the system. The exit can be at several terminators.



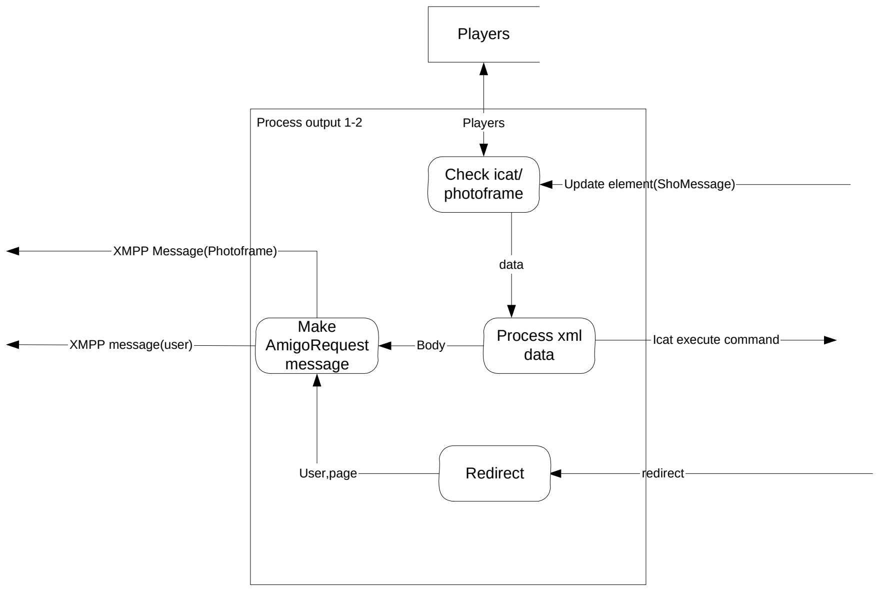
4.2.2 Running

The central game process contains all intelligence of the game. Output will be generated here. And input will be processed. And in the end there will be a winner.


StartGame
Players
Startgame signal
Running 1-2
Player
Next Player
signal
Process output 1-2
Update element(ShoMessage)
Callbacks(answer)
Ask Category
Icat event answer
Icat event handling
Category
Get Question from xml file
Question xml
Signal next
Update element (ShowMessage)
Callbacks(answer)
Ask question
Icat event answer
Answer
Update element (ShowMessage) result
Process answer
DataFlow1
Check winner
Winner

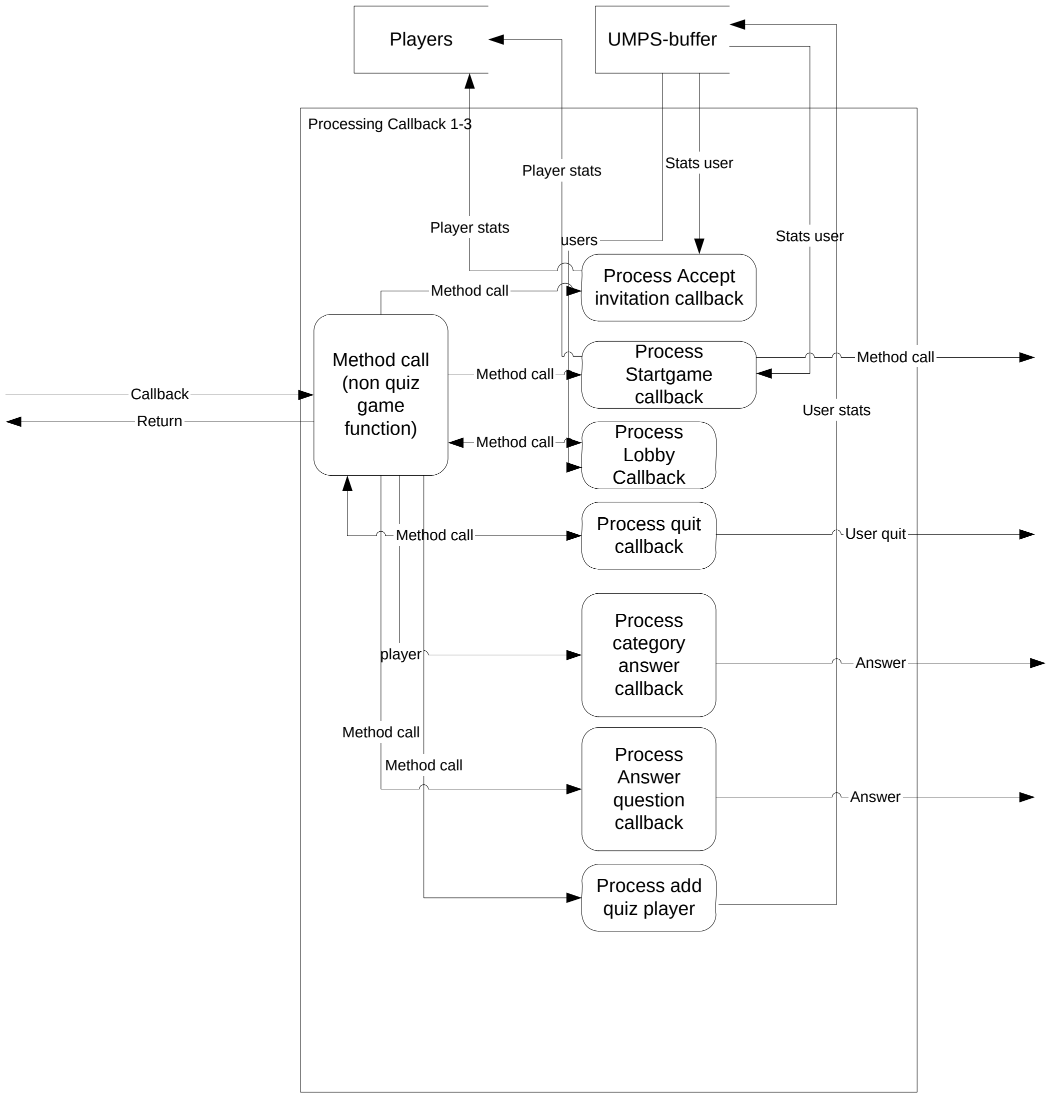
4.2.3 Process output

To send the output of the quiz in the right way to the user this process was created. All output that must be send to the users goes trough this process. In this way the developer does not care about how the output will be send. The most important thing is, it will be send.



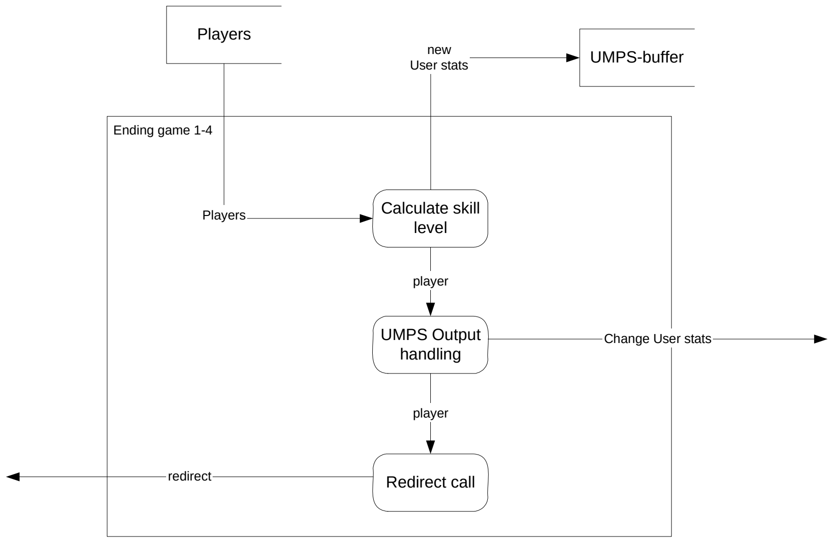
4.2.4 Process Call-Backs

This process is the router of the incoming messages.

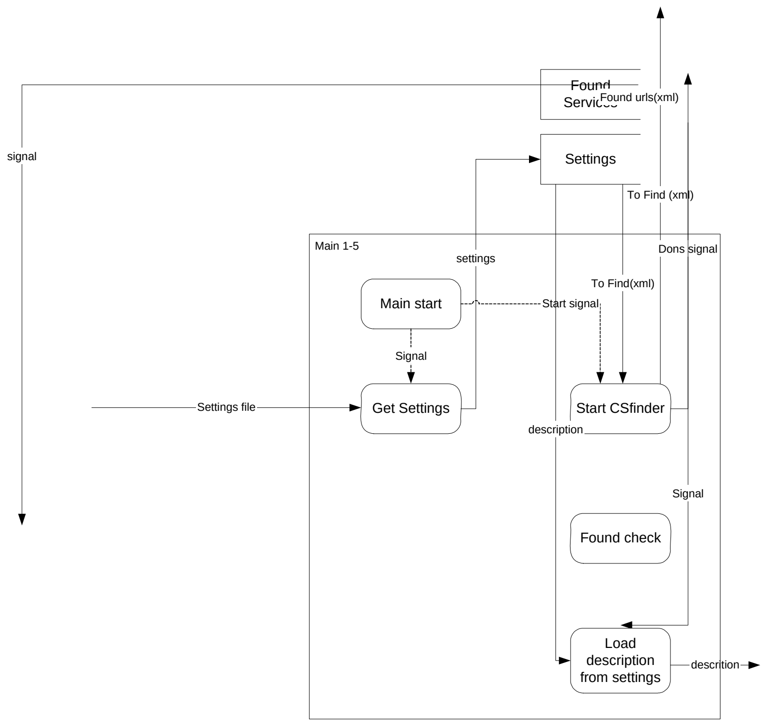


4.2.5 End game

Like the most games in the end there is a winner this diagram shows what will be don when there is a winner.

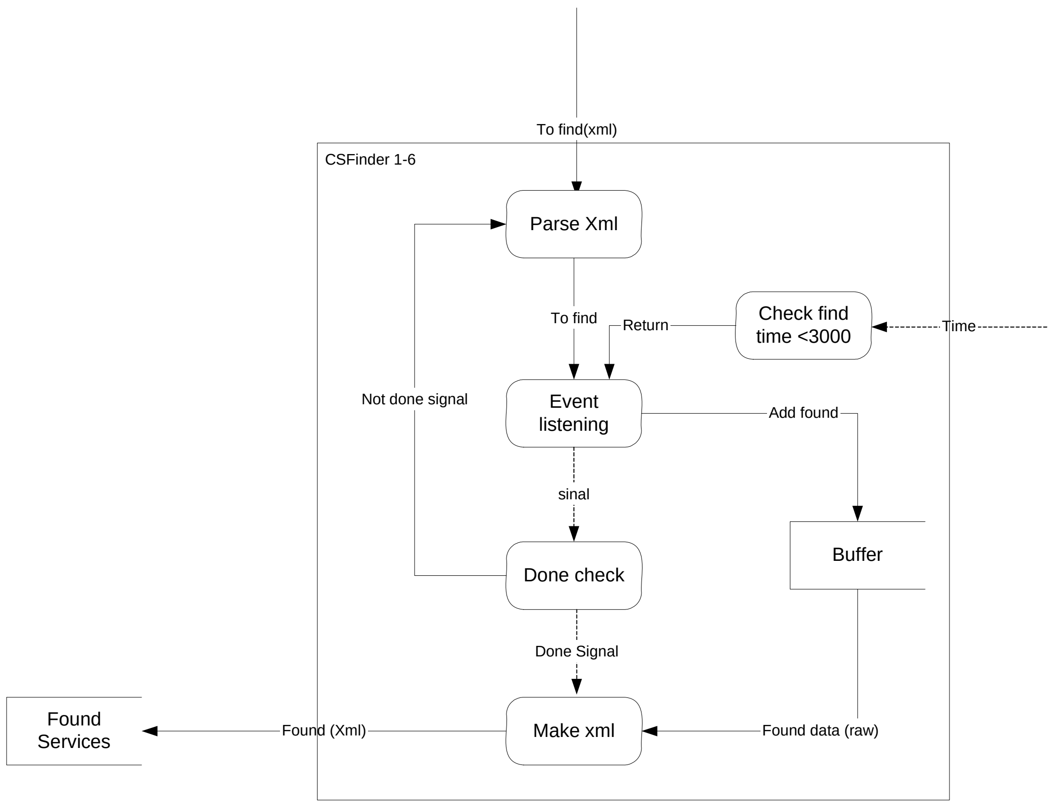


4.2.6 Main



4.2.7 CsFinder

Discovering other services, that is where it is all about. Incoming xml describes what should be found. And going out also xml with the found services.



4.2.8 Available Devices Publisher

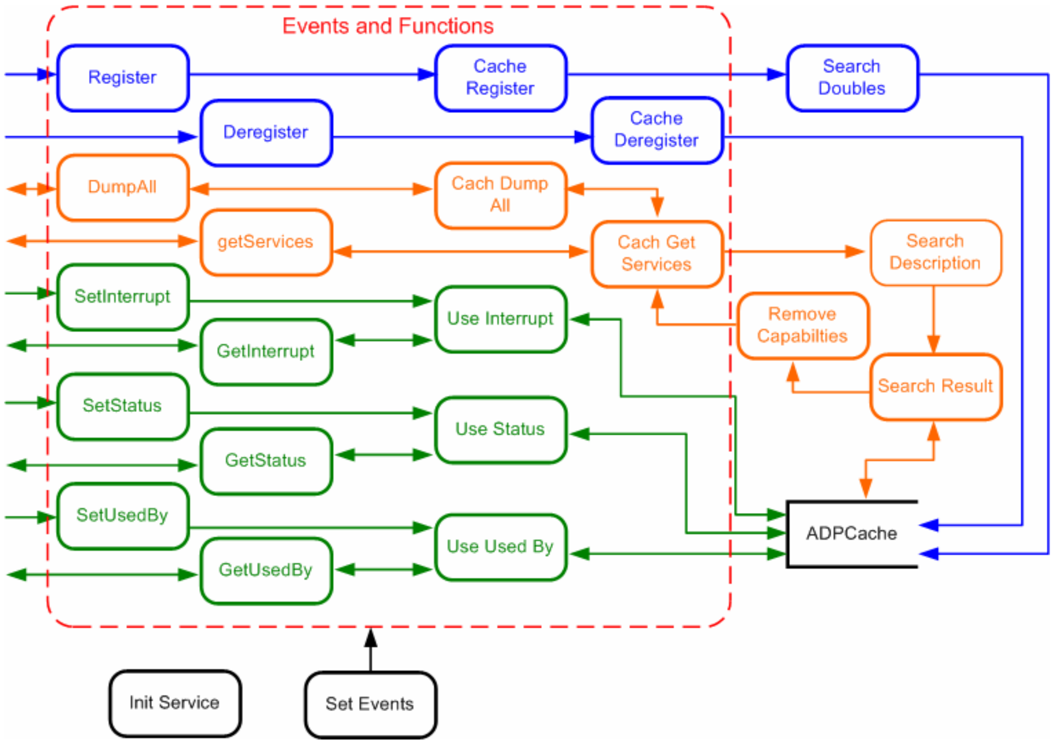


Figure 4-4 DFD Available Devices Publisher

The available devices publisher is used for several things. One thing is registering, deregistering services/devices so that other services can locate them. (Coloured blue in the diagram) Another is the ability to search for a specific service/device depending on the service description. (Coloured orange in the diagram). The last ability is to be able to set and get the status of service, status of interrupt and who it is being used by (Coloured green in the diagram).

Init Service
Initializes the service as a discoverable web service and publishes the functions of the service to the outside world.

SetEvents
Ties all incoming events of the web service functions to the real internal functions. Applies to all functions within the red box, on the left are the web service functions, on the right are the internal function bound to them.

ADPCache
XML Storage containing all device descriptions and capabilities

Register
Webmethod where the service can register it self here and pass its service description.

Deregister
Webmethod where the service can deregister it self here and pass its service description.

DumpAll
Webmethod that is used for debug of service and other services. Dumps all registered services with total description back to the caller.

GetServices
Webmethod that is able to search the cache with a specific device description. Returns the result in an XML format.

SetInterrupt
Webmethod to set the current interrupt status of the service/device at the ADP. Not needed to provide an entire service description. Only name and interrupt status in XML

GetInterrupt
Webmethod to get the current interrupt status of the service/device at the ADP. Not needed to provide an entire service description. Only name and interrupt status in XML

SetStatus
Webmethod to set the current status of the service/device at the ADP. Not needed to provide an entire service description. Only name and status in XML

GetStatus
Webmethod to get the current status of the service/device at the ADP. Not needed to provide an entire service description. Only name and status in XML

SetUsedBy
Webmethod to set the current usage of the service/device at the ADP. Not needed to provide an entire service description. Only name and usage name in XML

GetUsedBy
Webmethod to get the current usage of the service/device at the ADP. Not needed to provide an entire service description. Only name and usage in XML.

Cache Register
Internal function to register a service. Uses Search doubles to check if the cache already contains this service. If this checks out, registers the service.

Cache Deregister
Internal function to deregister a service. Checks if the service is present in the cache and the removes it from the cache.

Search Doubles
Searched the cache to see if the service description is already present in the cache.

Cache Dump All
Internal function that retrieves all services from the cache via the normal get services method, but does not filter anything and returns the entire list.

Cache Get Services
Internal function to get a specific service from the cache. This is done depending on the service description, if the name “All” is used, all services are returned.

Search Description
Analyze the service description to locate the specific search strings to process in search result.

Search result
Searches the cache for the search strings found in search description. The service descriptions of the results are combined in an XML string.

Remove description
Removes the capabilities part of the description form the XML string, so that a clean XML is returned to the user.

Use interrupt
Used to handle the interrupt status of a service or device, sets or gets the status from the cache for the specific service.

Use status
Used to handle the status of a service or device, sets or gets the status from the cache for the specific service.

Use used by
Used to handle the usage name of a service or device, sets or gets the name from the cache for the specific service.

4.2.9 User Profiles handling (Pokergame)

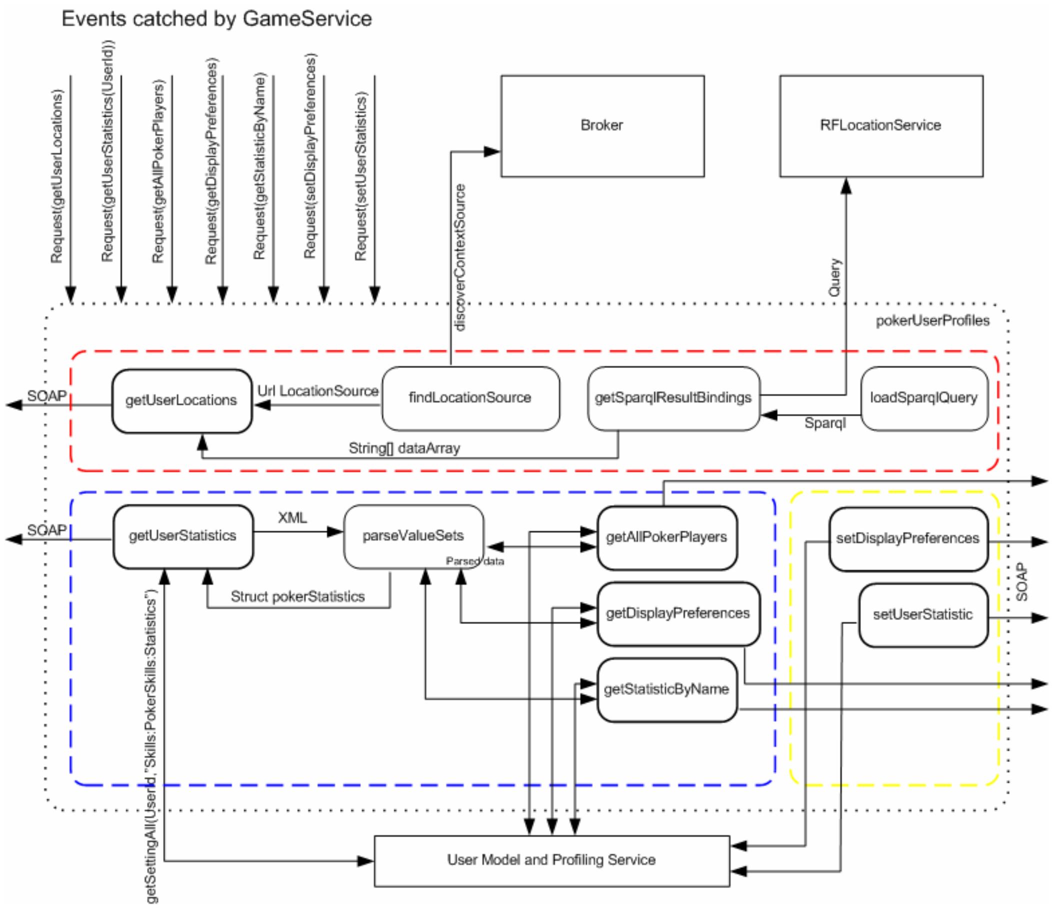


Figure 4-5 DFD User Profiles

The diagram above describes the class ‘pokerUserProfiles’. Three parts are separated by the dotted lines around it. The red part describes how the class gets the user locations in the house. The blue part describes all ‘get’ methods and the yellow part describes the ‘set’ methods. The bold boxes are public methods within the class and can be called by the Game Service.

GetUserLocations
To get the user location this method talks to the RFLocation ContextSource (See CMS developers guide). He will find this ContextSource by contacting the broker. When the method is called he will find the RFLocationService (findLocationSource) and returns an URL to the service. Now a query will be done at the RFLocationService in sparql ([RDQL]) language. The result from the RFLocationService will be parsed at the ‘getSparqlResultBindings’ and forwarded to the ‘getUserlocations’ as a string array containing users and locations.

GetUserStatistics
The public method ‘getUserStatistics’ will get all statistics from a ‘userid’ which will be send as parameter in this function. To get this information the method contacts the [UMPS] using SOAP mechanism. The return value of the [UMPS] is an XML based answer and will be parsed by the method ‘parseValueSets’. ‘parseValueSets’ returns a struct with all statistics from the requested ‘userid’ and will be send back to sender by SOAP.

GetAllPokerPlayers
’GetAllPokerPlayers’ will return al list of user names, which have set ‘pokerPlayer‘ as Personality key in his/her profile. Return value will be parsed by ‘parseValueSets’ and send to sender.

GetDisplayPreferences
This method will be used to get preferred displays to play the poker game. More important than the preferences is the location of the player. Return value will be parsed by ‘parseValueSets’ and send to sender.

GetStatisticByName
Return value will be parsed by ‘parseValueSets’ send to sender.
To get a single statistic from a user the ‘getStatisticByName’ can be used. With a string ‘userid’ and a string ‘StatName’ as parameters a single statistic (integer) will be returned from the [UMPS]. Return value will be parsed by ‘parseValueSets’ and send to sender.

SetDisplayPreferences
The first time the user will be asked to fill in his preferred displays. These will be saved to his profile. A Boolean will be returned, true if successful.

SetUserStatistic
This method will be used to save/update user statistics to his/her profile. StatId and StatValue will be send as parameters. A Boolean will be returned, true if successful.

4.2.10 Output (Pokergame)

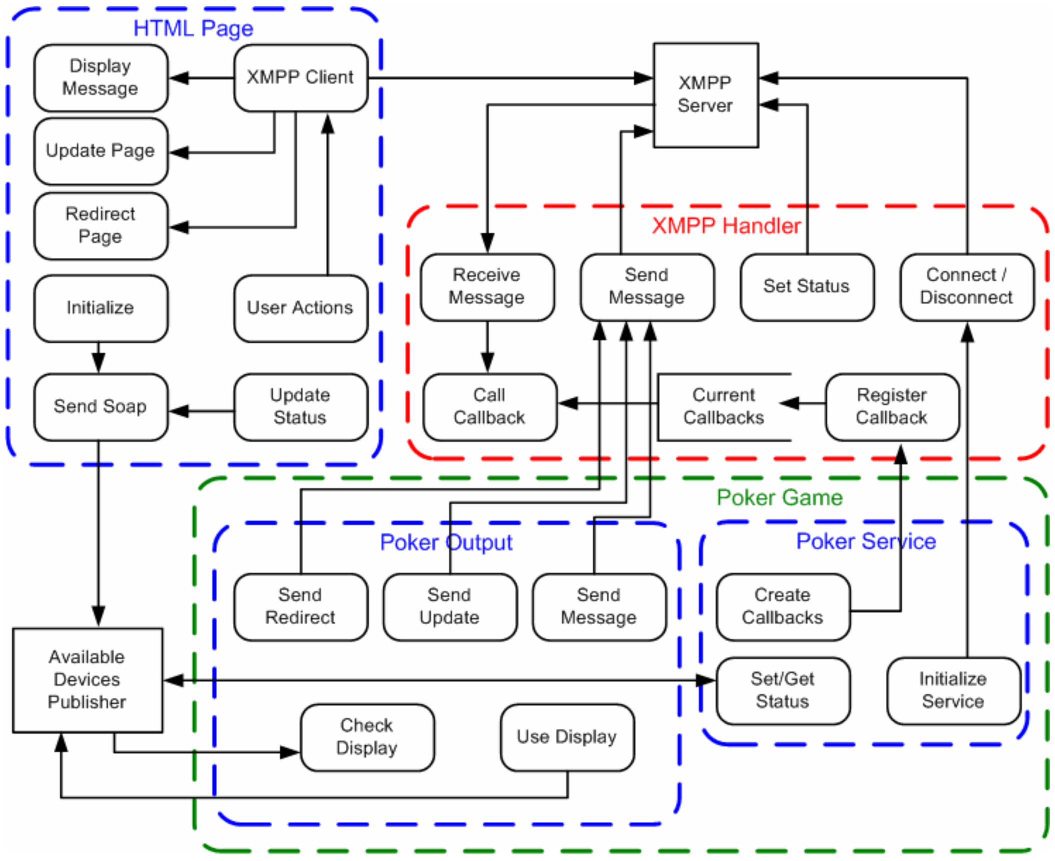


Figure 4-6 DFD Output

Available Device Publisher
The Available Device Publisher will be used to check for available displays, but also for the status part of the display and of the game service. The display can also set by which application or service it is being used.

XMPP Server
The XMPP / Jabber server is a service that can run anywhere, as long as all networks can reach it. The server regulated the registration and presence of users/services/devices that are logged in with a Jabber login. It ensures that messages are passed to the correct user and that all messages are send individually.

Connect / Disconnect
Connect or disconnect the current user, that was set at construction of the component, at the XMPP server. The user is not registered or removed, only online or offline status is set with this.

Register Callback
Register all callbacks at the XMPP Handler, the callbacks are from multiple classes inside of the poker service. This will be the game, the user profiles and every other class that needs an input from the user interface.

Current Callbacks
This is the storage of all currently registered callback functions that the Handler needs to pay attention to.

Set Status
Used to set the status and the status message of the user at the XMPP server.

Send Message
The generic function used to send messages to another XMPP user. The messages will be in an XML format specified in <link>.

Receive Message
Automatic message receive function. Called when message is received, analyzes the message and calls the call callback of the message type it was send with.

Call Callback
Calls the callback function that was registered with this specific message type when a message is received.

Initialize Service
The poker services is initialized and automatically registered with the XMPP server and connected.

Create Callbacks
All callbacks of all classes in the poker game will be registered at the beginning so callbacks will be able to commence from the start.

Set / Get Status
The status of the poker game can be set at the ADP, this status will contain info about the game status, such as “in game” or “in lobby”

Send redirect
When the display must be send to a new page, from login to lobby for example, this function will be called to redirect a specific user on the XMPP server to a new page url. Function is called via the XMPP send message function.

Send update
Send an update message to a specific display, a part of the screen must be updated. Which part of the screen will be passed on and also the new information will be passed.

Send message
Send a message / popup / overlay screen to a user or screen. This will contain information about the form of the screen and of the content of the screen.

Check display
Check display will retrieve information on the preferred screen and the status of the screen from the ADP. This information is checked, if this checks out the display is set using use display

Use display
The display that has been checked by check display is set as the current screen of this user at the ADP and in the user profiles. The screen is also redirected to the current game page and the status is accordingly.

Initialize
When the browser is activated a default page will be shown, this page will register the browser to the ADP as CE-HTML capable display device with a status available. This message will be to the web method on the ADP so a SOAP message will be used to call this.

Send SOAP
This generic part is able to send a SOAP message to a web method with certain content. Used for communication to ADP and other service except the game service.

Update Status
The status of the display will be set according to the page that is loaded, the active page will use this function to set the status at the ADP

XMPP Client
The XMPP client is active component within the Firefox browser that polls for data on the XMPP message connection. When a message is received it is passed further in the page and to other function that can process the specific message.

User actions
This is used for input from the user, when the user presses a button for example, the functions will pass this to the XMPP client so it will be send to the game service.

Display message
Here the message from the game service will be analyzed and put into a graphical part to display in the page

Update page
Here the update page message from the game service will be processed. The specified page part will be updated with the information. This must first be translated into CE-HTML components.

Redirect page
Here the message from the game service containing a page redirect will be analyzed and processed. The page will be loaded in the part of the browser where the application data may be put.

4.3 Static structure

4.3.1 Overview Class Diagram Quizgame

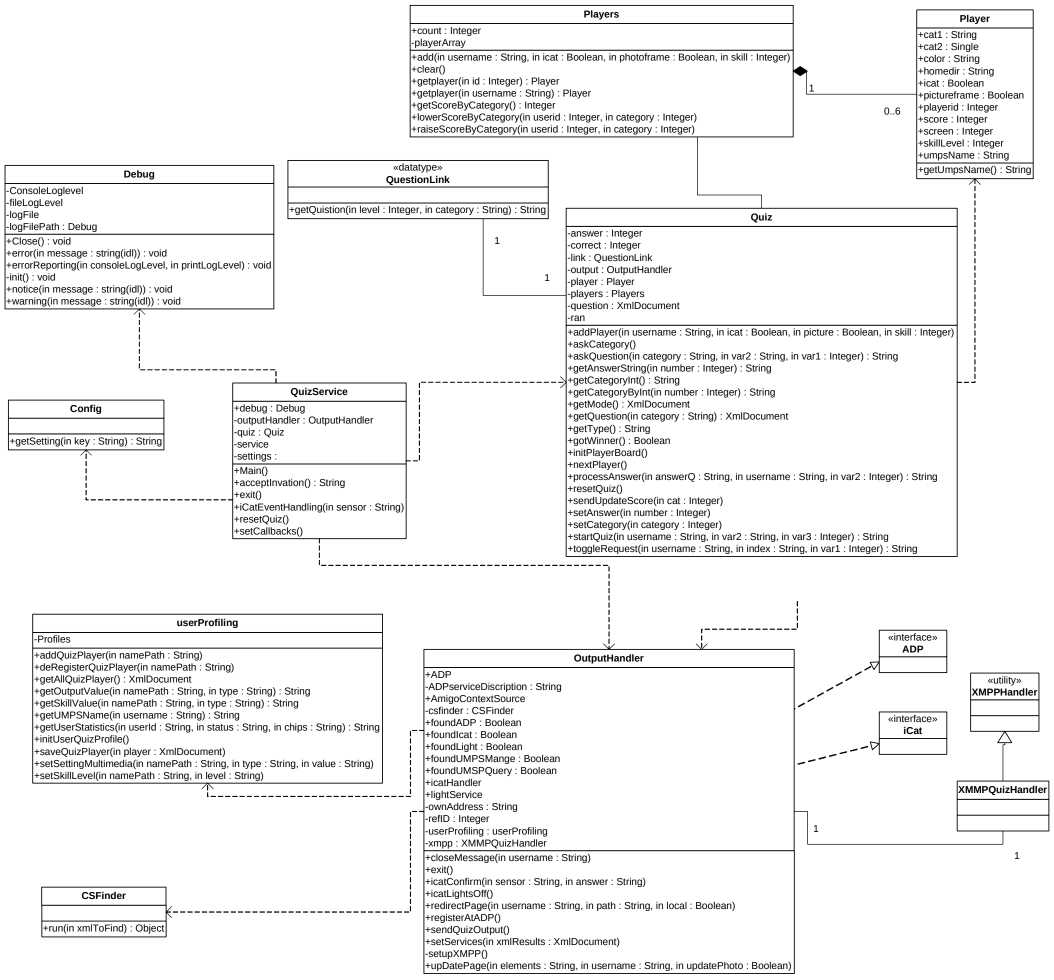

Players
+count : Integer
-playerArray
+add(in username : String, in icat : Boolean, in photoframe : Boolean, in skill : Integer)
+clear()
+getplayer(in id : Integer) : Player
+getplayer(in username : String) : Player
+getScoreByCategory() : Integer
+lowerScoreByCategory(in userid : Integer, in category : Integer)
+raiseScoreByCategory(in userid : Integer, in category : Integer)
1
0..6
Player
+cat1 : String
+cat2 : Single
+color : String
+homedir : String
+icat : Boolean
+pictureframe : Boolean
+playerid : Integer
+score : Integer
+screen : Integer
+skillLevel : Integer
+umpsName : String
+getUmpsName() : String
Debug
-ConsoleLoglevel
-fileLogLevel
-logFile
-logFilePath : Debug
+Close() : void
+error(in message : string(idl)) : void
+errorReporting(in consoleLogLevel, in printLogLevel) : void
-init() : void
+notice(in message : string(idl)) : void
+warning(in message : string(idl)) : void
«datatype»
QuestionLink
+getQuistion(in level : Integer, in category : String) : String
1
1
Quiz
-answer : Integer
-correct : Integer
-link : QuestionLink
-output : OutputHandler
-player : Player
-players : Players
-question : XmlDocument
-ran
+addPlayer(in username : String, in icat : Boolean, in picture : Boolean, in skill : Integer)
+askCategory()
+askQuestion(in category : String, in var2 : String, in var1 : Integer) : String
+getAnswerString(in number : Integer) : String
+getCategoryInt() : String
+getCategoryByInt(in number : Integer) : String
+getMode() : XmlDocument
+getQuestion(in category : String) : XmlDocument
+getType() : String
+gotWinner() : Boolean
+initPlayerBoard()
+nextPlayer()
+processAnswer(in answerQ : String, in username : String, in var2 : Integer) : String
+resetQuiz()
+sendUpdateScore(in cat : Integer)
+setAnswer(in number : Integer)
+setCategory(in category : Integer)
+startQuiz(in username : String, in var2 : String, in var3 : Integer) : String
+toggleRequest(in username : String, in index : String, in var1 : Integer) : String
Config
+getSetting(in key : String) : String
QuizService
+debug : Debug
-outputHandler : OutputHandler
-quiz : Quiz
-service
-settings :
+Main()
+acceptInvation() : String
+exit()
+iCatEventHandling(in sensor : String)
+resetQuiz()
+setCallbacks()
userProfiling
-Profiles
+addQuizPlayer(in namePath : String)
+deRegisterQuizPlayer(in namePath : String)
+getAllQuizPlayer() : XmlDocument
+getOutputValue(in namePath : String, in type : String) : String
+getSkillValue(in namePath : String, in type : String) : String
+getUMPSName(in username : String) : String
+getUserStatistics(in userId : String, in status : String, in chips : String) : String
+initUserQuizProfile()
+saveQuizPlayer(in player : XmlDocument)
+setSettingMultimedia(in namePath : String, in type : String, in value : String)
+setSkillLevel(in namePath : String, in level : String)
OutputHandler
+ADP
-ADPserviceDiscription : String
+AmigoContextSource
-csfinder : CSFinder
+foundADP : Boolean
+foundIcat : Boolean
+foundLight : Boolean
+foundUMPSMange : Boolean
+foundUMSPQuery : Boolean
+icatHandler
+lightService
-ownAddress : String
-refID : Integer
-userProfiling : userProfiling
-xmpp : XMMPQuizHandler
+closeMessage(in username : String)
+exit()
+icatConfirm(in sensor : String, in answer : String)
+icatLightsOff()
+redirectPage(in username : String, in path : String, in local : Boolean)
+registerAtADP()
+sendQuizOutput()
+setServices(in xmlResults : XmlDocument)
-setupXMPP()
+upDatePage(in elements : String, in username : String, in updatePhoto : Boolean)
«interface»
ADP
«interface»
iCat
«utility»
XMPPHandler
XMMPQuizHandler
1
1
CSFinder
+run(in xmlToFind) : Object

Figure 4-7 Class Diagram Overall

This diagram shows the overall view of the quiz service. All public classes and all methods are show, including interfaces. The relation between classes can be seen here, for specific explanation about a certain class, look at the paragraph of that class.

Config Library

This library contains the configuration off the game. All settings will be read from an xml file. The layout of this file will be described later in this document. The found settings can be get with the getSetting function. For example:

```
config = new Config("config.xml");
try
{
    Console.writeLine(config.getSetting("loglevel"))
}
catch (Exception e)
{
   Console.WriteLine(e.Message);
}
```

CSFinder Library

The CSFinder library contains a function to search sources in the amigo network. All services that are needed to start the game have to be set in the configuration file. For more information about this xml structure see the xml paragraph later in this document. The return value of the function will also be described in the xml paragraph.

Debug Library

The debug library will be created to handle error messages on a different level. The class can be initialised with many custom settings. These give the possibility of disable or enable messages by editing the configuration file.

Player

All in game users will be stored within this class. Also their attributes can be called from this class

Players

This is a container class for player storing. This means that an object of the class player will be stored here

Quiz

This is the class that contains the game logic. It also is some kind of gateway between player and output.

Outputhandler

This class contains all output handlers and output logic. It also holds the attribute of the userprofiling class.

Userprofiling

All user profiling between the game and the UMPS service takes place in this class.

Gameservice

This class contains the static main function as startup. It also contains the debug and setting class and some kind of simple menu.

QuestionLink

With this class the game can call some random questions fro the questionDB.xml file in the directory.

XMPP Handler Library

Equally as the pokeruserprofiles we will use the xmpp handler of the poker game. For detailed information watch the SDD document of the pokergame.

4.3.2 Overview Class Diagram Pokergame

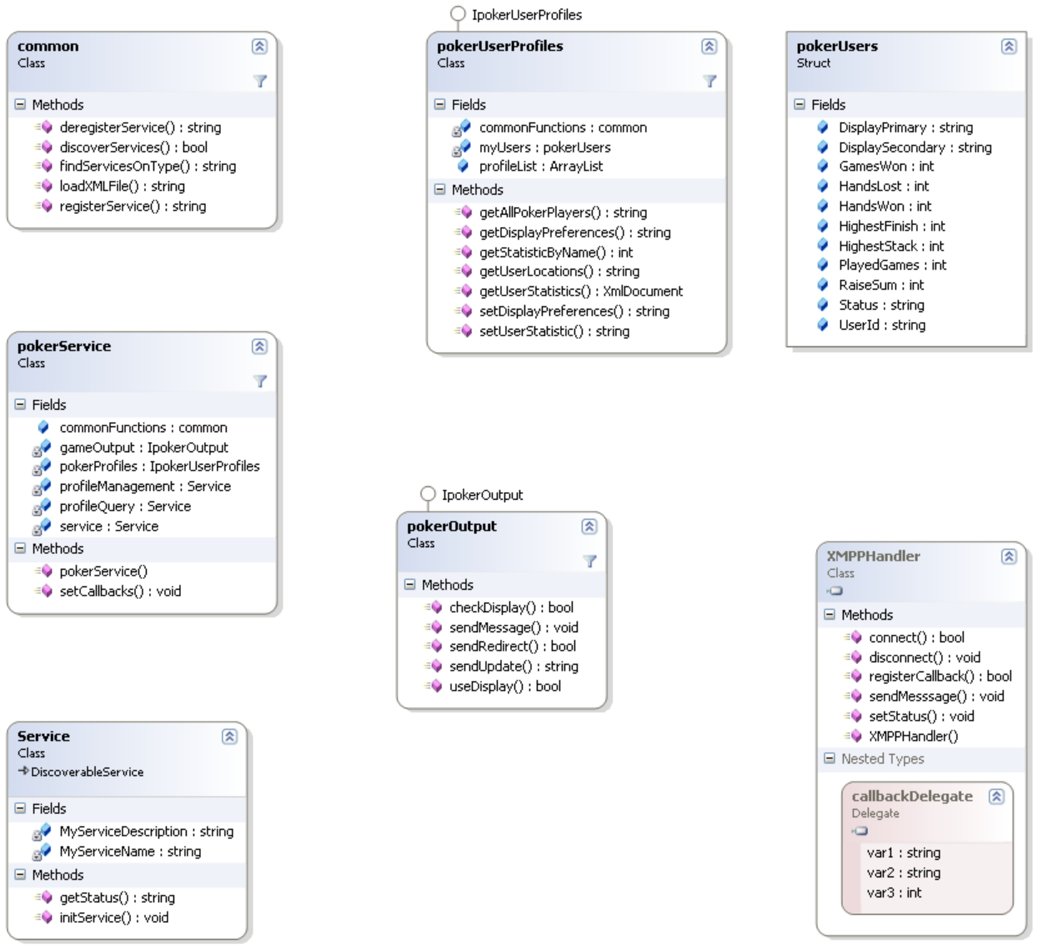
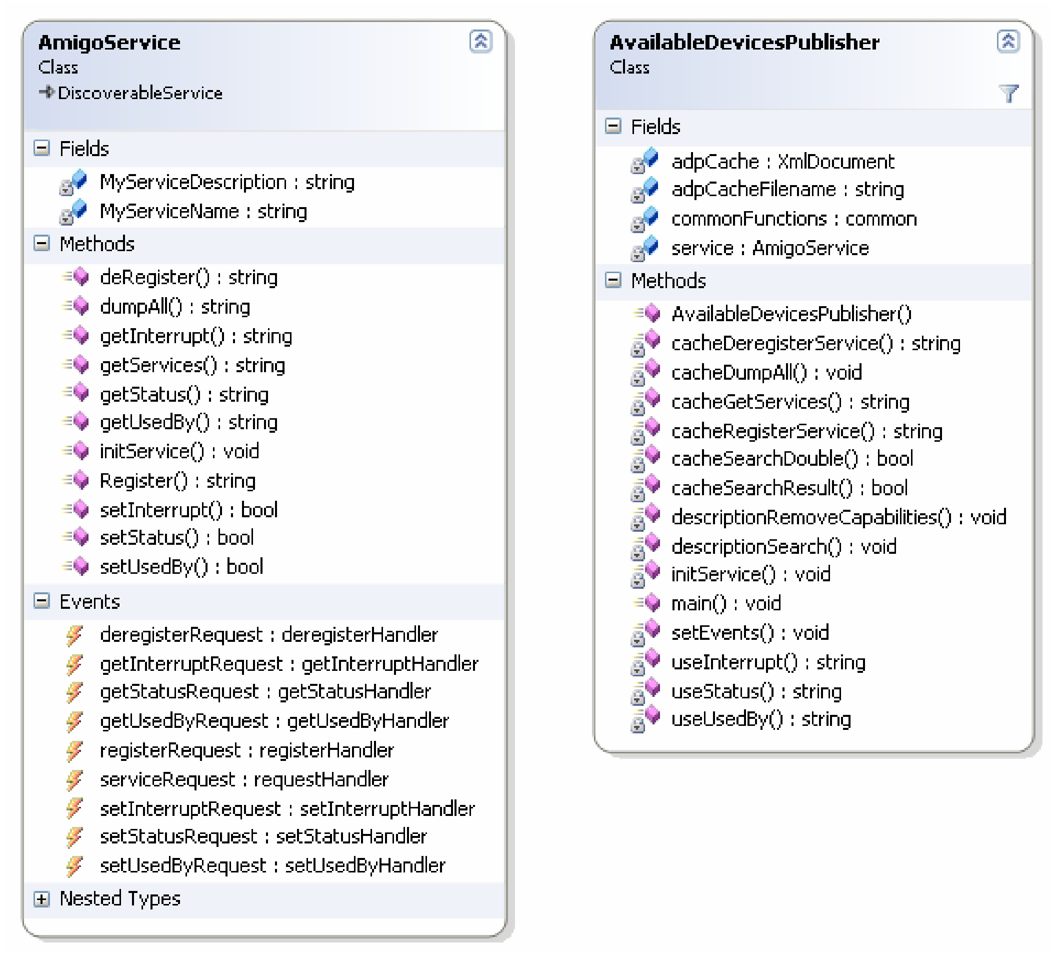


Figure 4-8 Class Diagram Overall

This diagram shows the overall view of the poker service. All public classes and methods are show, including interfaces. The relation between classes can be seen here, for specific explanation about a certain class, look at the paragraph of that class.

4.3.3 Available Devices Publisher



```
AmigoService
Class
→ DiscoverableService

Fields
    MyServiceDescription : string
    MyServiceName : string
Methods
    deRegister() : string
    dumpAll() : string
    getInterrupt() : string
    getServices() : string
    getStatus() : string
    getUsedBy() : string
    initService() : void
    Register() : string
    setInterrupt() : bool
    setStatus() : bool
    setUsedBy() : bool
Events
    deregisterRequest : deregisterHandler
    getInterruptRequest : getInterruptHandler
    getStatusRequest : getStatusHandler
    getUsedByRequest : getUsedByHandler
    registerRequest : registerHandler
    serviceRequest : requestHandler
    setInterruptRequest : setInterruptHandler
    setStatusRequest : setStatusHandler
    setUsedByRequest : setUsedByHandler
Nested Types
```

```
AvailableDevicesPublisher
Class

Fields
    adpCache : XmlDocument
    adpCacheFilename : string
    commonFunctions : common
    service : AmigoService
Methods
    AvailableDevicesPublisher()
    cacheDeregisterService() : string
    cacheDumpAll() : void
    cacheGetServices() : string
    cacheRegisterService() : string
    cacheSearchDouble() : bool
    cacheSearchResult() : bool
    descriptionRemoveCapabilities() : void
    descriptionSearch() : void
    initService() : void
    main() : void
    setEvents() : void
    useInterrupt() : string
    useStatus() : string
    useUsedBy() : string
```

Figure 4-9 Class Diagram [ADP]

The Available Devices Publisher contains two classes, one class for the web service component and one class for all internal functions.

Class: AmigoService Functions	Explanation:
All Events	These events are called when the web method is activated, this way the delegate function bound to them are called. Following this the functions in the AvailableDevicePublisher are called.
Deregister	Webmethod that calls the deregister event and returns the value
DumpAll	Webmethod that creates a generic description and the calls the getservices event and returns the value.
GetInterrupt	Webmethod that calls the getinterrupt event and returns the value
GetServices	Webmethod that calls the getservices event and returns the value
GetStatus	Webmethod that calls the get status event and returns the value
GetUsedBy	Webmethod that calls the getusedby event and returns the value

Class: AmigoService Functions	Explanation:
InitService	Initializes the function, sets the name and description of the service
Register	Webmethod that calls the register event and returns the value
SetInterrupt	Webmethod that calls the setinterrupt event and returns the value
SetStatus	Webmethod that calls the setstatus event and returns the value
SetUsedBy	Webmethod that calls the setusedby event and returns the value

Class: AmigoService Functions	Explanation:
CacheDeRegisterService	Searches the XML ADPCache for this service, if it exists removes the service from the cache.
CacheDumpAll	Called for the generic description with name: “All”. Retrieve all services from the cache en returns them in a XML format without any filtering
CacheGetServices	Used to search the cache for specific services. Uses *DescriptionSearch* to determine the precise search string. Then searches the cache with *CacheSearchResult* and filters the result with *DescriptionRemoveCapabilities*. Return the result in XML format.
CacheRegisterService	Checks the ADPCache if the service already exists with *CacheSearchDouble* and adds the services XML description to the cache if it is not yet present.
CacheSearchDouble	Searches the ADPCache if a service with the specific description already exists. Search will be based on service name.
CacheSearchResult	Searches the cache for a specific service based on three search strings. The XML node type, that node’s value and the parent of that node. Returns all result that match in an XML format.
DescriptionRemoveCapabilties	Filters the entered XML to a result XML. All capabilities are removed from the description. This way only useful information remains
DescriptionSearch	Searches the RDF description for the three search strings needed to search the cache.
InitService	Initializes the web service, making it discoverable and accessible.
Main	Main function, that ensures that the AvailableDevicesPubliser runs continuously.
SetEvents	Couples all internal function to the events in the service class. All function must comply with the standard of the specific delegate.
UseInterrupt	Sets or gets the Interrupt status of a specific service in the cache.
UseStatus	Sets or gets the status of a specific service in the cache.
UseUsedBy	Sets or gets the usage of user of a specific service in the cache.

4.3.4 Poker Output

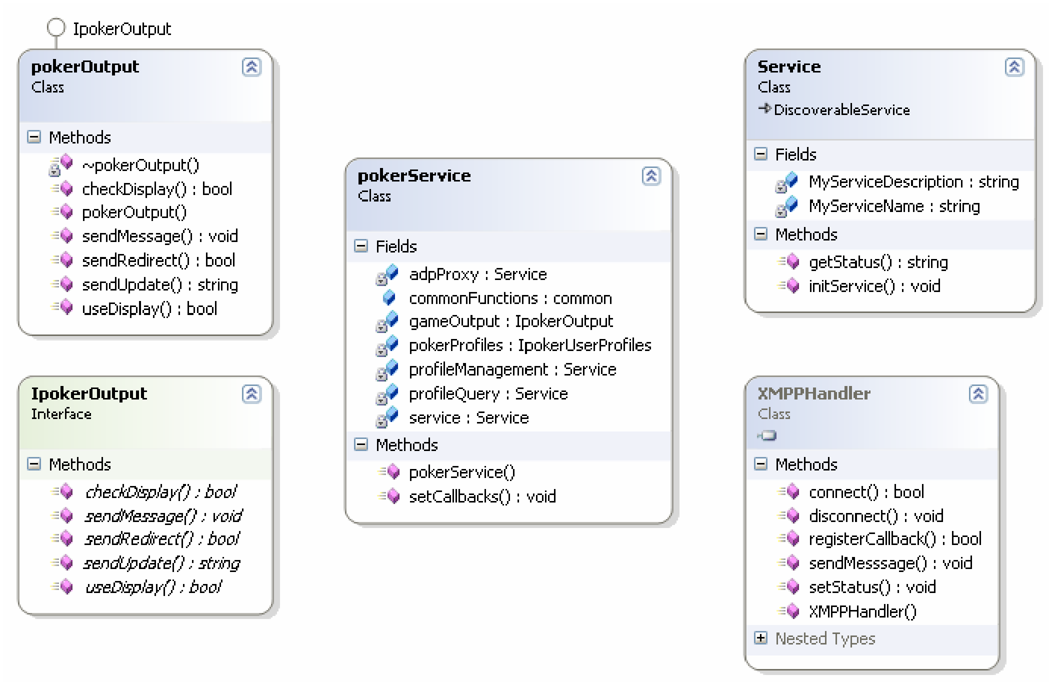


Figure 4-10 Class Diagram Output

This diagram shows the overall classes used in combination with the pokerOutput class. The pokerOutput class has interface available that should be used in the pokerService class. For the communication to and from the screen a XMPP Handler library will be used that makes connection to the displays and sends and receives messages. Also see *0 Config Library*

This library contains the configuration off the game. All settings will be read from an xml file. The layout of this file will be described later in this document. The found settings can be get with the getSetting function. For example:

```
config = new Config("config.xml");
try
{
    Console.writeLine(config.getSetting("loglevel"))
}
catch (Exception e)
{
   Console.WriteLine(e.Message);
}
```

CSFinder Library

The CSFinder library contains a function to search sources in the amigo network. All services that are needed to start the game have to be set in the configuration file. For more information about this xml structure see the xml paragraph later in this document. The return value of the function will also be described in the xml paragraph.

Debug Library

The debug library will be created to handle error messages on a different level. The class can be initialised with many custom settings. These give the possibility of disable or enable messages by editing the configuration file.

Player

All in game users will be stored within this class. Also their attributes can be called from this class

Players

This is a container class for player storing. This means that an object of the class player will be stored here

Quiz

This is the class that contains the game logic. It also is some kind of gateway between player and output.

Outputhandler

This class contains all output handlers and output logic. It also holds the attribute of the userprofiling class.

Userprofiling

All user profiling between the game and the UMPS service takes place in this class.

Gameservice

This class contains the static main function as startup. It also contains the debug and setting class and some kind of simple menu.

QuestionLink

With this class the game can call some random questions fro the questionDB.xml file in the directory.

XMPP Handler Library for this. The following classes are used, the functions are explained beneath:

Class: PokerOutput Functions	Explanation:
CheckDisplay	Retrieved display preferences from the user profile and compare these with the current status and availability at the ADP. Returns true if display is available, false if not available.
SendMessage	Constructs and sends an XML message prompt with text and message form to the display using the connected XMPP Handler
SendRedirect	Constructs and sends an XML redirect message with the redirect url to the display using the connected XMPP Handler.
SendUpdate	Constructs and sends an XML update message containing the part of the page to update and the update data to the display using the connected XMPP Handler.
UseDisplay	After device is checked, uses device for the player in this poker game. Device is set in poker game, user profile as active for this user. Device

Class: PokerOutput Functions	Explanation:
	status and usedBy is also changed in the ADP. Device is also set to the current game page.

4.3.5 XMPP Handler Library

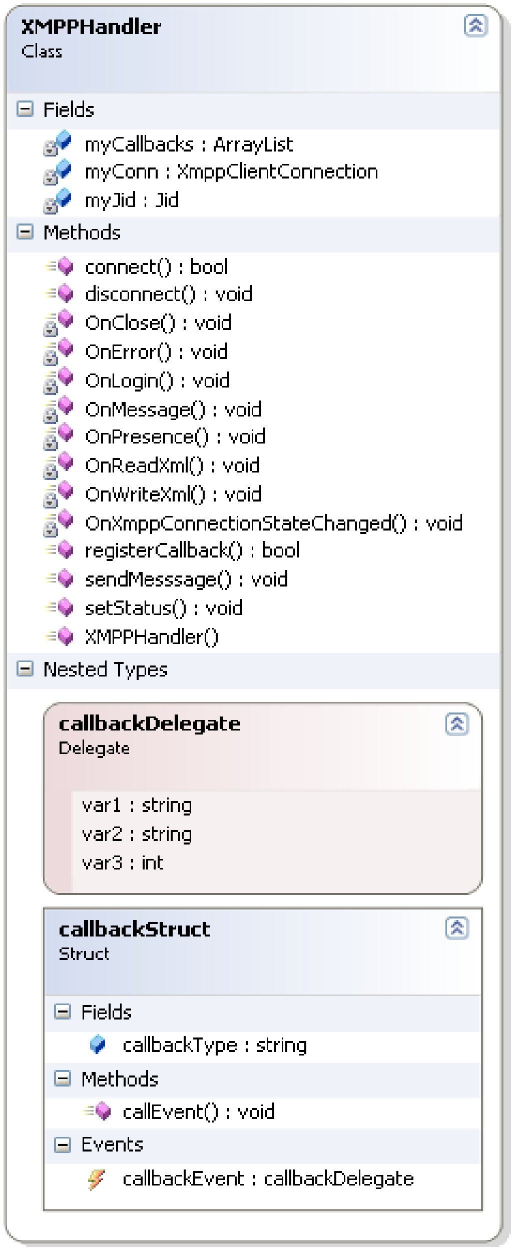


Figure 4-11 Class Diagram XMPP Handler

For the communication to the display we will use the XMPP protocol. For this an extension for the poker game has to be made. To make this part reusable for other amigo components this is done in a Dynamically Linked Library to include in other projects. This DLL works with callback functions. Each callback that can be registered, must be comply with the delegate function and have a specific messageType to handle. When a specific messageType is received the registered callback will be called automatically. The library can also be used to send messages in an XML format using the sendMessage function and set the status with the setStatus function.

Class: PokerOutput	
Functions	Explanation:
Connect	Used to connect to an XMPP server with a certain username and password. This data is passed with the constructor, and connection is opened using this function.
Disconnect	Closes the current open connection.
OnClose	Handler for event given on connection close. To handle success message.
OnError	Handler for event given on connection error. To handle any errors.
OnLogin	Handler for event given on connection login. To handle success message
OnMessage	Handler for event given on new message, main function to analyze the messages on message type and forward the message to the correct callback and via this way also the function in the poker Service.
OnPresence	Handler for event given on presence/status change. To set status of users.
OnReadXML	Handler for event given on each read request. Used for debug.
OnWriteXML	Handler for event given on each write request. Used for debug.
OnXMPPConnectionStateChanged	Handler for event given on connection state changed. To handle connection error in debug.
RegisterCallback	To register callback methods, give a pointer to the callback delegate method and the messageType it will be used for.
SendMessage	Send a message to a specific user on a server, can only be the server that the user is connected to.
SetStatus	Set the status and the status message of the current user.

CallbackStruct	
Functions	Explanation:
CallbackType	The string containing the message type for which this callback will be used, message type defined in XML structure in <link to format>.
CallEvent	Function to call the event, due to event security properties.
CallbackEvent	The event that triggers the callback delegate.

4.3.6 Javascript Interface (CHESS Interface)

On the interface side, 3 javascript components have been developed that can be used by a GUI to facilitate communication to jabber server and amigo services. These components are

Jabber

Contains functionality to use the notifsocket for setting up communication

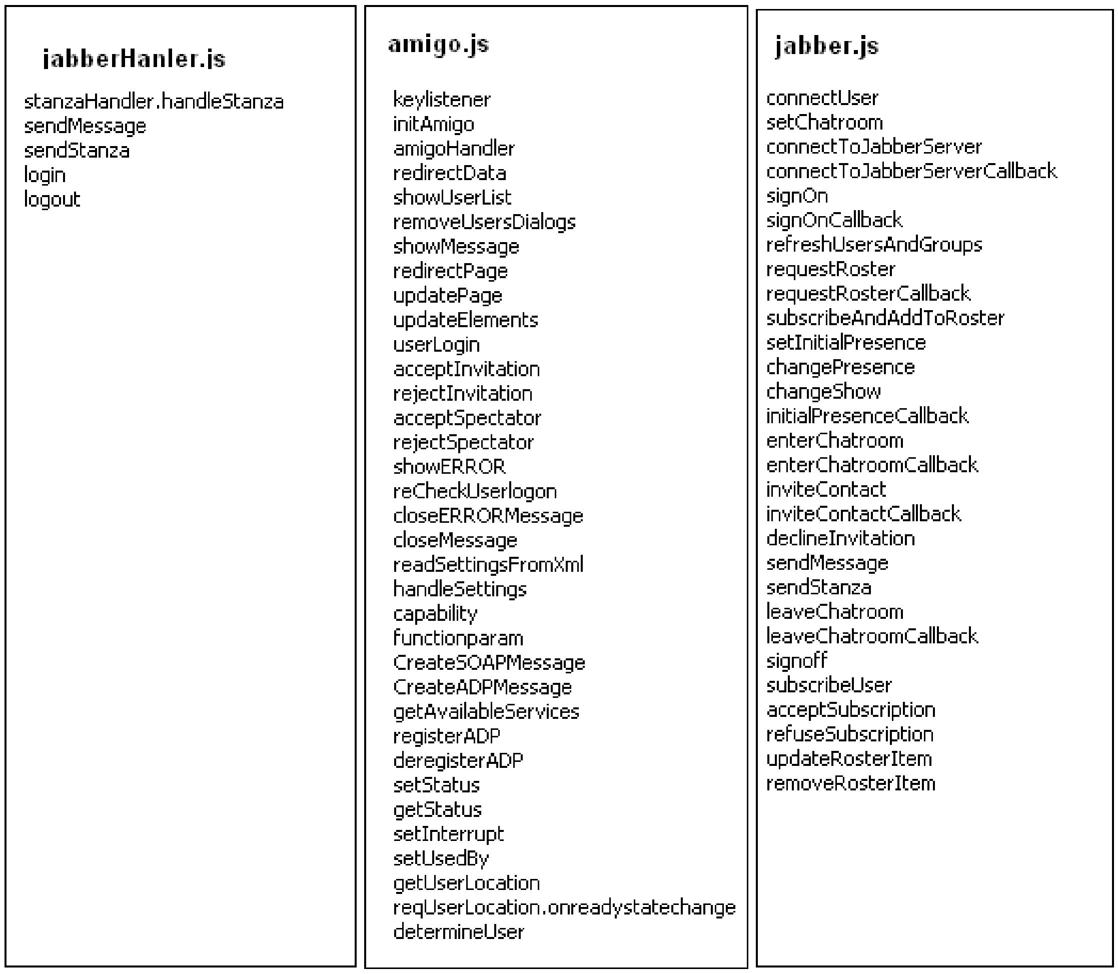
Jabberhandler

Contains functionality to collect and parse incoming (Amigo) messages

Amigo

Contains functionality to access Amigo services and to handle Amigo messages

In Figure 4-12 an overview of the functions in each file is presented



jabberHanler.js	amigo.js	jabber.js
stanzaHandler.handleStanza	keylistener	connectUser
sendMessage	initAmigo	setChatroom
sendStanza	amigoHandler	connectToJabberServer
login	redirectData	connectToJabberServerCallback
logout	showUserList	signOn
	removeUsersDialogs	signOnCallback
	showMessage	refreshUsersAndGroups
	redirectPage	requestRoster
	updatePage	requestRosterCallback
	updateElements	subscribeAndAddToRoster
	userLogin	setInitialPresence
	acceptInvitation	changePresence
	rejectInvitation	changeShow
	acceptSpectator	initialPresenceCallback
	rejectSpectator	enterChatroom
	showERROR	enterChatroomCallback
	reCheckUserlogon	inviteContact
	closeERRORMessage	inviteContactCallback
	closeMessage	declineInvitation
	readSettingsFromXml	sendMessage
	handleSettings	sendStanza
	capability	leaveChatroom
	functionparam	leaveChatroomCallback
	CreateSOAPMessage	signoff
	CreateADPMessage	subscribeUser
	getAvailableServices	acceptSubscription
	registerADP	refuseSubscription
	deregisterADP	updateRosterItem
	setStatus	removeRosterItem
	getStatus	
	setInterrupt	
	setUsedBy	
	getUserLocation	
	reqUserLocation.onreadystatechange	
	determineUser	

Figure 4-12 Overview of javascript functions

4.4 Data decomposition

4.4.1 SOAP Message Structure

SOAP messages will be used to get and post data from and to Amigo services. Via the XMLHttpRequests soap message are constructed and send. The XML scheme below shows the standard message.

```
<?xml                    version="1.0"                       encoding="utf-8"?>

<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <soap:Body>
    <query                                         xmlns="http://tempuri.org/">
      <contextQueryExpression>     SPARQL_QUERY     </contextQueryExpression>
    </query>
  </soap:Body>

</soap:Envelope>
```

4.4.2 Sparql generic function

To query the Amigo services the SOAP body messages will be filled with sparql queries.

The javascript function below can be used as a generic query function.

```
function       CreateSparql(subject,       subjecttype,       predicate,
reversepredicate,                   object,                   objecttype)
{
var                             sparqlquery                             =

'PREFIX  rdf:  &lt;http://www.w3.org/1999/02/22-rdf-syntax-ns#&gt;  '  +
'PREFIX   amigo:   &lt;http://amigo.gforge.inria.fr/owl/amigo#&gt;   '  +

'SELECT           ?subjectid           ?objectid           '           +

'WHERE {    ?context    amigo:'+    predicate    +'    ?object    .'    +
            '?context  amigo:'  +  reversepredicate  +  '  ?subject  .'  +
            '?subject     rdf:type     amigo:'+subjecttype+'     .'     +
            '?object     rdf:type     amigo:'+objecttype+'     .'     +
            '?subject     amigo:identifier     ?subjectid     .'     +
            '?object       amigo:identifier       ?objectid       }';
            <!—You          can          use          ORDER          BY-->
            <!-- Still  need  to  find  out  why  filter  does  not  work  -->
            <!--   "FILTER   ?personid   =   "+userinput+"   }";      -->

return                                                        sparqlquery;
}
```

4.4.3 Settings XML

This paragraph contains the structure of the settings xml file. The xml file MUST be named “config.xml”. All data in the xml file is dynamic and can be changed before the game service is started. It is not allowed to change the structure of the file. Otherwise the game will not start.

```
<?xml version="1.0" encoding="utf-8"?>

<settings>

     <setting>

       <settingvalue>

           <key>Name of the setting must be one word</key>

           <value>Value of the setting can contain xml</value>
```

```
              </settingvalue>
          </setting>
</settings>
```

4.4.4 CSFinder XML

The CSFinder library uses xml to read what service has to be searched. The structure of the parameter the run function uses is as following:

```
<amigo>
          <context>
              <type>Type of the service</type>
              <scope>Scope of the service</scope>
              <priority>Priority of return</priority>
          </context>
</amigo>
```

Using the priority tag give the user the possibility to order the findings. The findings of the CSFinder are returned in an xml document with the following structure.

```
<found>
          <context>
              <type>Type of the service</type>
              <scope>Scope of the service</scope>
              <address>service adress<address>
              <priority>Priority of return</priority>
          </context>
</found>
```

4.4.5 XML Communication

This is the structure of the XML format used in communication via XMPP to the browsers. The receiver side will call the function with the parameters if present. If not present a failed message must be returned.

```
<?xml version="1.0" encoding="utf-8"?>
<AmigoRequest>
          <From>user@server</From>
          <To>user@server</To>
          <Service>ServiceName</Service>
          <Function>FunctionName</Function>
          <Param0 name="param0Name" >Content</Param0>
          <Param1 name="param1Name" >Content</Param1>
          <Param2 name="param2Name" >Content</Param2>
```

```
</AmigoRequest>
```

The Icat handler also uses XML messages. This communication goes through a normal socket between the Icat handler and the other parts of the Icat interface. More details will be written in the Documentation of the service.

```
<icat>
     <type>Command type</Type>
     <command>Command or part</command>
     <param>Content</param>
     <frames>Content</frames>
</icat>
```

4.5 Design decisions

4.5.1 Design choices

Multi user

The multi user implementation can be don in a few different ways. In the following paragraph describes what that ways are. At the end of the paragraph will be described what way has been chosen.
The requirement in the [CRS(v1.0).doc paragraph 2.3.5] describes what should be possible by using multiple users. According to that paragraph are the following options for the implementation of this requirement.

Create multiple Notifsocket objects in the communication frame of the television menu. For each user that logs in one. This means that there will be a static limit of users that can be logged in at the same time. And possible there will be a limit of socket objects that can be created. The documentation of notifsocket is completed so these kinds of problems are not in it.

Create a data parser that redirects the data to the right user. This option uses only one notifsocket. There will be no limit of users that can log in at one television. Problem is other teams that use the television menu should also implement this option otherwise it won't work. And possible other demos will fail to run.

The solution that will be used will be the first. This will be the easiest way to log in two users. This will be enough for the moment.

It will only be possible to log in with two users at one television.

To implement the multi user requirement like described above. The following will be made. At the moment the JavaScript code is not Object oriented. This will be the first step that has to be made. After that it will be very easy to add a new user object so it is possible to login with two. The following diagram describes what will be made.

Testability

For the testing of the components debug messages will be printed to the default output of the program. This output will be standard in all the software components, this will not hinder the user. The output will contain simple messages about the steps of the service. However, if external components such as the broker contain bugs or has been released late, it might be needed to introduce extra debug functions. In that case they will be fully documented in this chapter.

4.6 Code organisation

4.6.1 Icat service

The Icat service will be placed in the folder Icat. This folder will have the following content:

`/`	Visual Studio Project Files and source code files
`/bin/`	Build output
`/obj/`	Compiler binary objects
`/Properties/`	Project configuration
`/Reference/`	Project Local Resources
`/Web Resources/`	Project Resources

4.6.2 Game Service

The Game Service will be placed in the folder QuizService. This folder will have the following content:

`/`	Visual Studio Project Files and source code files
`/bin/`	Build output
`/obj/`	Compiler binary objects
`/Properties/`	Project configuration
`/Web Resources/`	Project Resources

4.6.3 TelevisionMenu (Shared Interface)

The Television Menu will be placed in the folder TelevisionMenu. This folder will have the following content:

`/`	Index.html
`/_css/`	Contains all CSS files
`/_javascript/`	Contains all JavaScript files
`/_remote/`	Contains Remote files
`/images/`	All images

In the source directory the following files can be found:

Appendix

A. Amigo button test tool.

The Amigo button test tool is a small tool that was written especially for the Firefox browser from Mozilla. It creates a button in Firefox, using Xultu that can be used to add an html page as a 'service' in the Device regisitry.

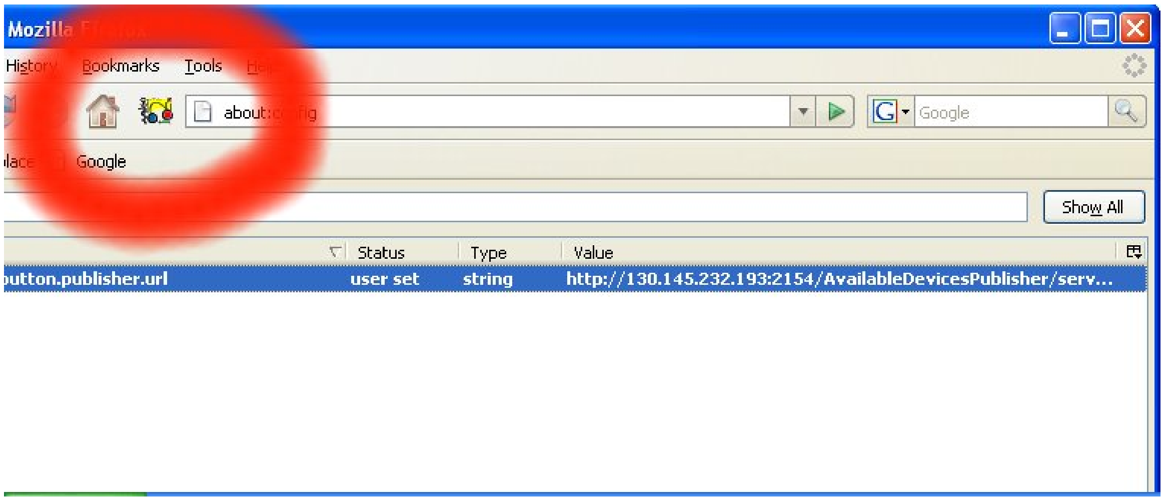
Configure

Tool is configured by setting the following preference in the about:config

extensions.amigobutton.publisher.url
http://130.145.232.193:2154/AvailableDevicesPublisher/service.asmx

Install

Copy the amigo button directory (completely) to the extensions directory of the firefox browser (the location of this directory differs for each version of firefox, please consult firefox documentation on how to install chrome applications)



	Status	Type	Value
utton.publisher.url	user set	string	http://130.145.232.193:2154/AvailableDevicesPublisher/serv...

Run

After installation, the button can be added to the firefox toolbar as shown above. When the button is pressed, the button will register the HTML page that is currently visited in the browser to the DevicesRegistry

Open tasks

When the link is to a local file then the tool assumes that this local file will be available on a local http server with same IP address as the publisher, it is the intention to have a configurable ftp address to copy the local files to. At this moment User will have to copy the files locally.

References

Ami05a	Amigo Deliverable D1.2: “Report on User Requirements”, M. D. Janse (ed.), IST-004182 Amigo, April 2005
Ami05b	Amigo Deliverable D2.1: “Specification of the abstract architecture of the Amigo System”, IST-004182 Amigo, 2005
Ami05c	Amigo Deliverable D2.2: “State of the Art”, Julia Kantorowitch (ed.), IST-004182 Amigo, 2005
Ami05d	Amigo Deliverable D2.3: “Specification of the Amigo Abstract System Architecture”, M.D. Janse (ed.), IST-004182 Amigo, July 2005.
Ami05e	Amigo Deliverable D3.1a: “Detailed Design of the Amigo Middleware Core; Service Modelling for Composability”, J. Kalaoja (ed.), IST-004182 Amigo, September 2005.
Ami05f	Amigo Deliverable D3.1b: “Detailed Design of the Amigo Middleware Core; Service Specification, Interoperable Middleware Core”, N. Georgantas (ed.), IST-004182 Amigo, September 2005.
Ami05g	Amigo Deliverable D4.1: “Report on Specification and Description of Interfaces and Services”, M.D. Janse (ed.), IST-004182 Amigo, October 2005.
Ami05h	Amigo Annex I – “Description of Work”, update T0+12-T0+30, September 2005
Ami06	Amigo Deliverable D3.2: Prototype implementation. N. Georgantas (ed.), IST-004182 Amigo, in progress.
Ami06b	Amigo Deliverable D4.2: Report on detailed Intelligent User Interface design, B Kladis (ed)
Ami06c	Amigo Deliverable D3.3
Men03	Diego Rios Mendoza, “Using Ontologies in Context-Aware Services Platforms”, Master Thesis, University of Twente, Enschede, The Netherlands, 2003.
CE-HTML	CEA-2014 - Web4CE / CE-HTML specification, 2006
XMPP	RFC3921 - XMPP specification