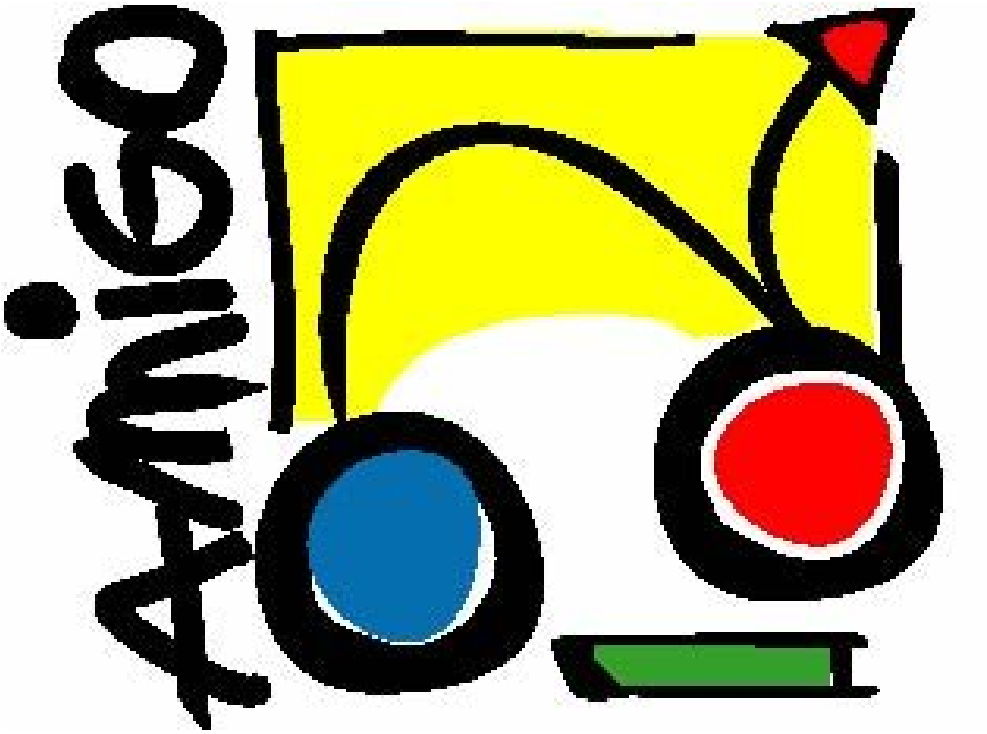

# IST Amigo Project
# Deliverable D3.4

## Amigo Overall Middleware: First Prototype Implementation & Documentation

First integrated methodology ('how to') for employing the middleware

IST-2004-004182
Public

Information Society Technologies

| | | |
|---|---|---|
| **Project Number** | : | IST-004182 |
| **Project Title** | : | Amigo |
| **Deliverable Type** | : | Report and Prototype |

| | | |
|---|---|---|
| **Deliverable Number** | : | D3.4 |
| **Title of Deliverable** | : | Amigo Overall Middleware: First Prototype Implementation & Documentation – First integrated methodology ('how to') for employing the middleware |
| **Nature of Deliverable** | : | Public |
| **Internal Document Number** | : | amigo_d3.4_correctedfinal |
| **Contractual Delivery Date** | : | 28 February 2007 |
| **Actual Delivery Date** | : | 21 May 2007 |
| **Contributing WPs** | : | WP3 |
| **Editor(s)** | : | **INRIA:** Graham Thomson, Nikolaos Georgantas |
| **Author(s)** | : | **FT:** Anne Gérodolle, Mathieu Vallée |
| | | **ICCS-NTUA:** Ioanna Roussaki, Dimitris Tsesmetzis, Yiannis Papaioannou, Miltiades Anagnostou |
| | | **IMS:** Edwin Naroska |
| | | **INRIA**: Graham Thomson, Sébastien Bianco, Nikolaos Georgantas, Sonia Ben Mokhtar, Valérie Issarny, Nicolas Palix, Charles Consel, Laurent Réveillère, Wilfried Jouve |
| | | **Microsoft:** Ron Mevissen, Stephan Tobies, Rich Hanbidge |
| | | **TELIN:** Pravin Pawar, Remco Poortinga |
| | | **TID:** José María Miranda, Álvaro Ramos, David Cordón, Andrés Tuells |
| | | **VTT:** Jarmo Kalaoja, Ilkka Niskanen, Toni Piirainen |


# Abstract

D3.4 concerns the first *overall* prototype implementation and associated documentation of the Amigo middleware. Specifically, we present in this document the first integrated methodology ('how to') for employing the Amigo middleware. D3.4 comprises: (i) the present document; (ii) developed source code of components; (iii) developed service description vocabulary and language ontologies; (iv) user's guide and developer's guide documents for components and ontologies; and (v) Javadoc-style and OWLDoc electronic documentation for components and ontologies. Delivered material besides the present document can be accessed on the Amigo OSS Repository - Public Web site (http://amigo.gforge.inria.fr/home/index.html). The present document, more specifically, proposes a set of "HOWTO" guides oriented towards the Amigo application and service developer. Together, these showcase the functionalities of the Amigo middleware, describing the options a developer has for creating Amigo services and

applications. The Amigo middleware is comprehensive, providing support for a large number of aspects of creating novel applications for the networked home environment. A HOWTO is provided for each approach for a particular aspect that the middleware supports. Each HOWTO highlights the features of a particular approach, and where alternatives exist, the relative advantages of each. In this way, the Amigo developer can use this document to assist in quickly choosing the best approach for the task at hand, and familiarizing themselves with how to use the approach in the development of their own Amigo services and applications.

## Keyword list

HOWTO, ambient intelligence, networked home system, interoperability, mobile / personal computing / consumer electronics / domotic domain, semantic concept, ontology, service description vocabulary, service description language, semantic reasoning, service matching, service composition, service adaptation, service execution, middleware, service discovery protocol, service interaction protocol, programming and deployment framework, context, quality of service, multimedia streaming, content distribution, security, data storage.

# Table of Contents

# Figures

# Tables

# 1 Introduction

This deliverable concerns the first *overall* prototype implementation and associated documentation of the Amigo Base Middleware (or simply middleware), after the two previous partial prototypes of Deliverables D3.2 [Amigo-D3.2] and D3.3 [Amigo-D3.3]. Specifically, we present in this document the first integrated methodology ('how to') for employing the Amigo middleware. We propose a set of "HOWTO" guides oriented towards the Amigo application and service developer. Together, these showcase the functionalities of the Amigo middleware, describing the options a developer has for creating Amigo services and applications. Our HOWTO guides are detailed enough to give a comprehensive overview of all the capabilities of the middleware offered to the developer and the basic steps for using these capabilities. They further reference other more detailed sources – in particular, the user's guide and developer's guide documents accompanying each middleware component, which are discussed below – where the developer can find component specifications and thorough examples of use. Further, this document includes a first qualitative assessment of the Amigo middleware, where we state the added value of each of the proposed middleware functionalities for the application developer. Actually, this is the first version of a living HOWTO document for the Amigo middleware, which will be updated based on feedback from the ongoing integration of the middleware into the application work packages WP5, 6, 7.

This document is complemented by the actual Amigo middleware prototype and its online documentation, where we follow the reporting model and update the material of the previous deliverables D3.2 and D3.3:

- On the Amigo OSS Repository - Public Web Site [Amigo-OSS-Pub] (see [Amigo-D9.5]), for each component under development:
    - Source code of the current prototype version;
    - User's guide and developer's guide documents, if already available;
    - Javadoc[1] (or equivalent for C#) documentation, if already available.
- On [Amigo-OSS-Pub], for the service description vocabulary and service description language:
    - OWL specification of the current version;
    - User's guide and developer's guide documents, if already available;
    - OWLDoc[2] (follows the same principle as Javadoc) documentation, if already available.

We note here that, with regard to component/ontology implementation and documentation, our focus during the last 6 months has been on completing the implementation of the components/ontologies; we have, thus, released complete versions for all of them. Consequently, the online documentation that we provide for certain components is still at an early stage and will take a form closer to complete when the almost final, public versions of components/ontologies will be available. Certainly, there are already a number of advanced user's guide and developer's guide documents for some of the components, which are specifically referenced by the present HOWTO document.

Introducing further the present document, we point out that it presents the comprehensive Amigo middleware, which provides support for a large number of aspects of creating novel

[1] http://java.sun.com/j2se/javadoc/

[2] http://www.co-ode.org/downloads/owldoc/co-ode-index.php

applications for the networked home environment. A HOWTO is provided for each approach for a particular aspect that the middleware supports. Each HOWTO highlights the features of a particular approach, and where alternatives exist, the relative advantages of each. In this way, the Amigo developer can use this deliverable to assist in quickly choosing the best approach for the task at hand, and familiarizing themselves with how to use the approach in the development of their own Amigo services and applications.

The first HOWTO of the deliverable, “How to develop a service” is presented in Chapter 2, and describes how the OSGi and .Net programming frameworks provided by the Amigo middleware allow a Java or C# developer to: make a Java or C# object remotely available as an “Amigo Service” (that is, a networked entity able to answer remote HTTP/SOAP calls, and act as an asynchronous event source); publish an Amigo service through WS-discovery; discover Amigo services using WS-discovery; and interact with discovered Amigo services, that is, place remote calls, subscribe to an event source and asynchronously receive events.

Then in Chapter 3, “How to develop a secure service”, a HOWTO is provided which informs the Amigo developer how to secure access to the services in the home.

In Chapter 4, “How to develop a semantic service”, HOWTOs are provided covering the range of options the Amigo developer has for describing semantic services, including simple, atomic capability descriptions, complex conversation-based capability descriptions, event-based service descriptions, as well as context-aware and quality-of-service descriptions. The use of the Amigo semantic service repository is also covered in this chapter.

Chapter 5, “How to develop a semantic service-based application”, provides HOWTOs that illustrate how to develop an Amigo application that exploits the suite of service discovery and service composition methods that the Amigo middleware offers.

There are a number of off-the-shelf domotic systems and devices based on technologies such as BDF, EIB, and X10, which can be installed at home but cannot be directly integrated within the Amigo platform. Chapter 6, “How to develop a domotic service”, describes the steps required to develop a domotic service, and how to integrate a domotic device into the Amigo system.

Chapter 7, “How to develop a multimedia content application”, walks the reader through building an application that uses the Amigo Middleware to manage and render multimedia content.

The Amigo Datastore is a service that implements a simple persistency layer for storing data within the Amigo networked home, and Chapter 8 provides a tutorial on how to use this service.

The last HOWTO of this deliverable is provided in Chapter 9, “How to use home system deployment, configuration and management”, which illustrates the use of the Amigo home management console, which provides the inhabitants/administrator of an Amigo home with a single point of contact regarding service control and diagnostics for the home’s networked environment.

Finally, Chapter 10 concludes this deliverable, summarizing the main features presented by these HOWTOs.

## 1.1 References


[Amigo-D3.2] Amigo Consortium. Deliverable D3.2: Amigo Middleware Core - Prototype Implementation & Documentation. March 2006.

[Amigo-D3.3] Amigo Consortium. Deliverable D3.3: Amigo Middleware Core Enhanced: Prototype Implementation & Documentation. October 2006.

[Amigo-D9.5] Amigo Consortium. Deliverable D9.5: Web site for sharing open source software developed within Amigo. March 2006.

[Amigo-OSS-Pub] Amigo Consortium. Amigo OSS Repository - Public Web Site. http://amigo.gforge.inria.fr/home/index.html

# 2 How to develop a basic service

## 2.1 Overview

### 2.1.1 Objectives and principles

Development of a basic service, i.e. a service that follows certain syntactic standards and has not (but may further be enriched with) a semantic description, is based on the Amigo programming and development framework.

A basic service may be a middleware (Base Middleware or Intelligent User) service or an application service.

The goal of the programming and deployment framework is two-fold:

- help developers programming Amigo services (or applications using Amigo services) without caring about underlying protocols, so as to reduce programming effort and enforce interoperability;
- maintain a repository of Amigo components that can be deployed on .Net or OSGi platforms.

The use of this framework is not mandatory, and developers may also package Amigo-aware services as independent applications that are to be deployed on a given system or hardware (as they see fit). Both kinds of components will be able to interact within the same Amigo environment through service discovery protocols, communication protocols and (when necessary) interoperability methods. However, the use of the framework reduces programming effort, eases reusability of components and enforces interoperability.

The programming framework is based on the following protocols:

- WS-Discovery for publishing and discovering Amigo services;
- HTTP/SOAP for remote method invocation;
- WS-Eventing for subscription to event sources.

We have made additional restrictions for the use of these protocols. For example, the WS-Eventing standard does not specify the transport protocol to be used for event delivery. The programming framework uses TCP. This is transparent to the developer. However, event sources or consumers developed outside the framework can interact with event sources or consumers based on the programming framework, provided they use TCP for event delivery.

### 2.1.2 Features

The OSGi and .Net frameworks allow a Java or C# developer to:

- make a Java or C# object remotely available as an “Amigo Service” (that is, a networked entity able to answer remote HTTP/SOAP calls, and act as an asynchronous event source);
- publish an Amigo service through WS-Discovery;
- discover Amigo services using WS-Discovery;
- interact with discovered Amigo services, that is, place remote calls, subscribe to an event source and asynchronously receive events.

A Java client may discover and interact with C# .Net services, and vice versa. However both frameworks have their own particularities. These particularities are detailed in the following sections; here are the main differences:

| *Feature* | *.Net framework* | *OSGi framework* |
|---|---|---|
| • Deployment platform | • .Net | • OSGi |
| • Operating system | • Windows (XP, CE, …) | • Windows(XP, CE, ...), Linux, Mac OS |
| • Integration in an IDE | • Yes (Visual) | • No |
| • WSDL handling | • Complete | • Incomplete |

*Table 2-1: A comparison of the features supported by the .NET and OSGi programming and deployment frameworks*

### 2.1.3 Assessment

The programming and deployment framework allows developers to program Amigo services (or applications using Amigo services) without caring about underlying protocols. The framework handles the protocols for announcement and discovery of remote services (WS-Discovery), synchronous interaction (HTTP/SOAP) and subscription to asynchronous sources of events (WS-Eventing).

Using this framework limits the problems of incompatibility due to different interpretations of the protocol specifications.

Several components in Work Package 3 and other work packages are using this framework. This eases understanding among developers.

Management tools exist or will be built, so that it will be possible to install, manage and monitor in a unified way, components that are deployed over the .Net or the OSGi framework.

## 2.2 How to develop a service with the .Net programming and deployment framework

### 2.2.1 Principles and prerequisites

The .Net programming framework is composed of several .Net libraries on top of the standard Microsoft .Net and .NetCF (.Net Compact Framework) platforms that all together enable application developers to build service-oriented components without having to deal with protocol details or complex structures. These libraries offer additional discovery, deployment and eventing mechanisms, that are not available in the standard .Net and .NetCF platforms, based on Web Service standards (resp. WS-Discovery, Web Services, and WS-Eventing).

The .Net platform is used on desktop, laptop and tablet PCs whereas the .Net Compact Framework is used on PDAs and SmartPhones.

The .Net or .NetCF runtime (available from http://download.microsoft.com) is required for these components at runtime. The .Net or .NetCF SDK is required at development time (also available from the link above). Although the components are written using C#, the binary form is a standard .Net library and can therefore be used with any .Net (e.g., Visual Basic .Net or the free Visual Studio Express editions) compliant language/tool.

### 2.2.2 How to develop, deploy and use services

A service-oriented architecture is always composed of a client and a server component. As such, the .Net libraries offer functionality for both sides (as far as the standard Microsoft .Net platform doesn't already support it; an example is the client side of a web service that is natively supported by .Net).

The latest version can be downloaded from the public Gforge website: http://amigo.gforge.inria.fr/. After installation the following directory structure is created:

\Program Files\EMIC\EMIC WSDiscovery Framework

- the main directory containing the license and the online documentation

\Program Files\EMIC\EMIC WSDiscovery Framework\Code

- the main directory for the source, binaries and example code

\Program Files\EMIC\EMIC WSDiscovery Framework\Code\Final

- the components (in binary form for .Net and .Net Compact Framework) that a developer **should** use when using the .Net programming framework

\Program Files\EMIC\EMIC WSDiscovery Framework\Code\Picture Frame Sample

- the example source code of a picture frame application (client and server)

\Program Files\EMIC\EMIC WSDiscovery Framework\Code\WSDiscovery

- the source code for the .Net components

\Program Files\EMIC\EMIC WSDiscovery Framework\Code\Dependencies

- the components necessary for the sample to run

#### 2.2.2.1 Discovery, deployment and eventing terminology and libraries

A service-oriented architecture is composed of at least one server and one client component. The Amigo project selected Web Services as the interoperability protocol, thus developers need to write (and deploy) a Web Service. This is enabled by the EMIC.WebServerComponent.dll (this can be found in the 'Final' directory discussed in the previous section). This component is used for the server side. A client will use Web Services to call methods on the server. There is no additional component necessary since this is natively supported by the .Net platform (http://msdn.microsoft.com/webservices)

Web Services are hosted on devices and there might be many of them available in an Amigo system. Locating a Web Service (that is, getting their network address) is solved by discovery. This is enabled (for client as well as server side) by the EMIC.WSDiscovery.dll (also in the 'Final' directory).

Eventing is a mechanism that allows components to be notified over the network with signals. These signals are called events and can potentially carry data with them (e.g.: the location of a user changed, where the new location is the data carried by the event). Eventing is enabled by the EMIC.WSEventing DLL.

#### 2.2.2.2 Sample server

(Note: a full working example can be found in the 'Code' directory after installing the EMIC Programming Framework).

A simple server (one that returns the current date/time) requires a class that implements the server component (sample implementation in C#):

```
public class MyFirstService
{
   [WebMethod]
   public DateTime GetDateTime()
   {
      return DateTime.Now;
   }
}
```

(For details, please read the Web Service introduction at http://msdn.microsoft.com/webservices)

This class exposes a single method as a Web Service method (GetDateTime()).

The first step to deploy this class as a Web Service is to register an instance of this class with the EMIC.WebServerComponent DLL:

```
WebServer.Port = 8080;
WebServer.AddWebService("MyFirstService", new MyFirstService());
```

The first line instructs the WebServer to use port 8080 (0 (zero) could also be used as the port, in which case the WebServer will chose an arbitrary free port).

The second line creates a new instance of the MyFirstService class and registers it with the WebServer component under the virtual directory "MyFirstService". Note: multiple services can be registered with a single WebServer as long as they all use different virtual directories.

The web service is now registered with the following address: http://<machinename>:8080/MyFirstService/service.asmx

#### 2.2.2.3 Sample client

For the client, we rely on the standard .Net functionality of generating client stubs (proxies) for Web Services. This can be done using the wsdl.exe tool or directly from the Visual Studio IDE (by adding a web reference to the address above after which proxy code is generated automatically).

#### 2.2.2.4 Sample server with discovery

Adding discovery (making a service discoverable) using the .Net Programming Framework is shown below: it uses the server code from above and adds the following extensions (shown in boldface) making the service discoverable.

```
[Location("MyLocation")]
[Endpoint("MyEndPoint")]
[Type("MyType")]
[Scope("MyScope")]
public class MyFirstService : DiscoverableService
{
        [WebMethod]
        public DateTime GetDateTime()
        {
                return DateTime.Now;
        }
}
```

The location, endpoint, type and scope attribute are metadata that are used in the discovery process and are fully detailed in the online documentation.

The DiscoverableService is a base object that takes all complexity away from the developer and handles WS-Discovery automatically. The announcement (Hello and Bye in WS-Discovery) are connected to the object's lifetime so that a Hello message is sent when the object is instantiated and a Bye when the object is disposed.

#### 2.2.2.5 Sample client with discovery

The .Net programming framework also has support for client side discovery. There are two ways of locating a service with WS-Discovery. The first is by scope and type and is called a Probe. A query is sent to the network and all services implementing this scope and type answer (refer to the scope and type attribute demonstrated in the code snippet of the previous section). The second one is by Endpoint reference (again, refer to the code snippet of the previous section, but this time for the Endpoint attribute) and is called Resolve. An Endpoint is a unique reference to a specific service (instance), hence only that service will answer.

Two objects, a ProbeClient and a ResolveClient can be used for the two previously described query methods. Their API is very similar:

```
ProbeClient pc = new ProbeClient();
pc.AddServiceScope(new ServiceScope("MyScope"));
pc.AddServiceType(new ServiceType("MyType"));
pc.SendProbe();
Thread.Sleep(3000);
Console.WriteLine("I found "+pc.ProbeMatches.Count" services.");
```

The first line instantiates a ProbeClient object followed by two lines that add the query requirements: the Scope and the Type we are looking for. The fourth line sends out the probe over the network. The sleep of 3 seconds (3000 milliseconds) allows the probes to arrive. The answers to the probe can be received through the ProbeMatches property of the ProbeClient object.

Note: there is also an asynchronous way to get notified for every arriving answer, see the online documentation for further details.

An example of a resolve can be found is given below:

```
ResolveClient rc = new ResolveClient();
rc.EndpointReference = new EndpointReference("MyEndpoint");
rc.SendResolve();
Thread.Sleep(3000);
if (rc.ResolveMatches.Count != 0)
        Console.WriteLine("I found the service.");
```

Note: there is also an asynchronous way to get notified for every arriving answer, again see the online documentation for further details.

#### 2.2.2.6 Sample server with eventing (event source)

When using the .Net programming framework, eventing can be added to an existing Web Service (see section "Sample server" above). The mechanism used is standard C# events that are mapped to WS-Eventing events. The following example shows how this can be done:

```
[WSEvent("MyFirstEvent")]
public class MyFirstService
{
        public event EventHandler<WSNotificationEventArgs> MyFirstEvent;

        [WebMethod]
        public DateTime GetDateTime()
        {
                if (this.MyFirstEvent != null)                  // C# event signal
                        this.MyFirstEvent(this, null);          // (continued)

                return DateTime.Now;
        }
}
```

The WSEvent attribute tells the underlying deployment module (EMIC.WebServerComponent.dll) that the C# event with the name MyFirstEvent should be mapped to a WS-Eventing event. Now, every time the C# event is fired (see the commented lines), a WS-Eventing event is sent out to the network.
(Note: the check if the event is not null checks if there are any subscribers, there is an implicit

subscriber – the underlying deployment module – but if not, a null pointer exception would be thrown).

The code above is only for sample purposes (not functional): an event is fired every time somebody calls GetDateTime().

An event may carry additional data. See the online documentation for further details and example code.

#### 2.2.2.7 Sample client with eventing (event sink)

Events can be received using the WSEventSink object:

```
private void EventReceived(    string eventName,
                                        NotificationData notificationData,
                                        bool subscriptionEnded)
{
        Console.WriteLine("Received event");
}
…
WSEventingSink sink = new WSEventingSink(
        new Uri("http://<machinename>:8080/MyFirstService/event")))
sink.Subscribe(this.EventReceived);
```

The last three lines first create an instance of the WSEventSink object, and then register the EventReceived method to be called every time an event is received. In the example above, the address of the event source is hard coded. When combined with discovery, the address that is received from the Probe/ResolveClient can be used.

In the example above, the event sink (client) subscribes to all events from that source (server). There are additional methods to specify which events a sink wants to subscribe to. Besides this, subscriptions have an expiration period and need to be renewed. See the online documentation for more details and example code.

### 2.2.3 Resources

http://download.microsoft.com (frameworks, SDK)

http://msdn.microsoft.com/webservices (Web Service technology)

http://amigo.gforge.inria.fr/ (open source Amigo repository)

## 2.3 How to develop a service with the OSGi programming and deployment framework

### 2.3.1 Principles and prerequisites

The OSGi programming framework is mainly composed of a set of Java classes and interfaces (packed in OSGi bundles), and some tools used during development. The API is defined by one bundle called amigo_core, so that developers need only to include amigo_core.jar in their compilation environment. Other Amigo bundles are used only at run-time. Instances of classes implementing the interfaces defined by amigo_core are discovered through OSGi lookup.

#### 2.3.1.1 Development

Development of a bundle using the Amigo OSGi framework requires a Java Development Toolkit, at least JDK 1.3. It is further recommended to install ant from http://ant.apache.org.

#### 2.3.1.2 Run-time

At run-time, the components can be deployed on any CDC Java Virtual Machine, Personal profile or above. An OSGi R3 platform is required. The framework has been tested with Oscar 1.0.5 [oscar] and Knopflerfish [Knopflerfish]. Pre-configured versions of Oscar for J2SE on PC (Windows, MacOS, Linux), J2SE on a Linux PDA and J2ME/Personal profile on a Pocket PC can be downloaded from http://amigo.gforge.inria.fr/obr/. See the user's guide for more details.

### 2.3.2 How to develop, deploy, and use services

The Amigo OSGi programming framework is not an integrated programming framework. Java developers may use any IDE (Integrated Development Environment) like Eclipse or Netbeans, or no IDE at all. To allow a quick start, developers may download a template project from [amigo_sample]. This contains all the necessary resources (Java source, build files, libraries) to build OSGi bundles covering the following main features: creating an Amigo service and publishing it on the network; discovering and using an Amigo service; acting as an event source; subscribing to an event source and receiving events.

The "How-to" part of the developer's guide [amigo_dev] details these different features, which we rapidly review below. We distinguish between two use cases:

**Use case 1**. We are developing a server and a client from scratch in Java. We shall define our API in Java and make Amigo services from plain Java classes (POJO).

**Use case 2**. For the server, there already exists a WSDL description. We want to write a client that will connect to this server. This is typically the case if the service was written using the C# framework.

We describe only the case where we are developing a client and server from scratch. The second use case is detailed in the complete how-to, to be found in the developer's guide on the Amigo Gforge web site. Not covered here either is the writing of the OSGi manifest and possible metadata files.

#### 2.3.2.1 How to create an API bundle

First we create a bundle that will define the API of our Amigo services, and stub classes which will be used to access Amigo services. We design the API according to our needs. We have to use the "java2stub" tool to generate additional information and stub classes that will be packed together with the Java interfaces in an "API bundle".

#### 2.3.2.2 How to create a bundle that provides an Amigo service

The first task is to create a class that implements the API we have just defined. We have then to instantiate this class, create an "AmigoService" instance and register (if desired) this service on the network. This looks like:

```
server = new MyImplementation(…);
AmigoExportedService myService=serviceExporter.createService(server);
try {
    myService.exportInterface(AmigoReference.DEFAULT,Hello.class);
    logger.info("exported service with reference "+myService.getReference());
} catch (AmigoException e) {
    System.err.println("error exporting object",e);
}
```

The AmigoExportedService encapsulates any object to make it an AmigoService. exportInterface will expose the object on the network (using a communication protocol not decided here). In this example, only the methods defined by the Hello interface are exposed onto the network.

**Publishing an Amigo service**

We may (or may not) publish the service using WS-Discovery:

```
try {
    myService.addProperty("location",mylocation);
    myService.addProperty("serviceType","HelloService");
    lookup.registerService(myService);
} catch (AmigoException e) {
    e.printStackTrace();
}
```

The service will be published using the default scope urn:amigo, the service type "HelloService", and in this example an additional property called "location".

**Acting as an event source**

Once the service has been created, it can act as an event source. This is done by using the EventSender and NotificationData interfaces. Here is a code excerpt that retrieves a reference on the event sender associated to the Amigo service, creates an event containing two pieces of information (a location and a temperature) and sends it asynchronously.

```
final EventSender sender = myService.getEventSender();
NotificationData data = new NotificationData();
data.setProperty("location", myLocation);
data.setProperty("temperature", temperature);
sender.notify("helloEvent",data);
```

Calling the notify method puts the event in a queue to be asynchronously sent to all (if any) event consumers having subscribed to this event source for the "helloEvent" type (see "Subscribing to event sources and receiving events" below).

#### 2.3.2.3 How to create a bundle that uses an Amigo service

We have seen the basic principles to expose an Amigo service. Now let's see the "client" part, that is: discovering services, invoking methods, subscribing to event sources.

**Discovering an Amigo service**

Amigo services are discovered using the AmigoLdapLookup methods lookup or lookupFirstService, for example:

```
AmigoService[] services = lookup.lookup ("(serviceType=HelloService)");
```

**Invoking an Amigo service**

The AmigoService we have just discovered is only a placeholder for the end point reference of the service and some properties (e.g., the "location" property in the previous example). To place a remote call, we will build a stub that implements the required interface.

```
Hello helloService=(Hello) (services[0].getSpecificStub(Hello.class));
```

Then, invoking the helloWorld method on this stub will build a soap envelope containing the request, post it to the server, retrieve the answer and decode it.

```
helloService.helloWorld(someArg);
```

**Subscribing to event sources and receiving events**

To receive events, a client must provide an implementation of the EventSink interface:

```
public class MyEventSink implements EventSink {…
```

This interface defines two methods that we must implement:

```
public void subscriptionEnded(String tag){
   System.out.println("subscription ended");
}

public void notify(String tag,  NotificationData data){
   System.out.println("received an event");
   for (Enumeration keys=data.keys();keys.hasMoreElements();){
            String key = (String) keys.nextElement();
            System.out.println(key+":"+data.get(key));
  }
}
```

notify is called every time an event to which the event sink has subscribed is received. subscriptionEnded will be called when the subscription ends (on the sink unsubscription request, or on the source decision).

To receive events from an event source, we must create an instance of event sink and call the subscribe method:

```
EventSink sink=new MyEventSink();
service.getSubscriptionManager().subscribe (sink,"helloEvent",-1);
```

The -1 means that we have subscribed for ever to the event source, to receive events sent with the "helloEvent" tag. Reception of events is done in a separate thread, and the "notify" method of our event sink will be called each time an event is received from the source.

To unsubscribe from the event source (e.g. when the bundle is stopped) we call the unsubscribe method:

```
service.getSubscriptionManager().unsubscribe(sink);
```

**Variants**

There are several ways to obtain a stub to an Amigo service reference: lookup is only one of them. A service may also be known by configuration (the URL is known, hard-coded or read from a configuration file), or as the result of a remote method invocation of some bootstrapping object. In such cases, one may use the AmigoImportedService createService method:

```
AmigoReference ref = new AmigoReference(AmigoReference.SOAP,"http://.....");
AmigoService service = AmigoImportedService.createService(ref);
```

Also, generating code is not mandatory. A generic API is also available.

```
service.getGenericStub().invoke("helloWorld",parameterNames,params);
```

#### 2.3.2.4 How to deploy bundles

The OSGi bundles we have just created can be deployed on any R3 compliant platform. The user's guide [amigo_user] shows how to install the Oscar OSGi platform and install bundles from the Amigo Bundle Repository. Once we have installed the Amigo bundles (that is, the core API bundle, and the protocol management bundles, amigo_wsdiscovery and amigo_ksoap_export), we may install the bundles we created using for example the Oscar Graphical User Interface.

There are three main ways to deploy bundles on a platform:

- At bootstrap: we may configure the OSGi platform so as to start a given list of bundles when starting. In the case of Oscar, we should edit the system.properties file and change the oscar.auto.start.1 property to include all bundles we need. We may use this technique if we want to distribute an application composed of OSGi bundles but we do not want to exhibit the OSGi aspects.

- Installing a bundle from a given URL: Oscar provides textual, graphical or remote (through a browser) user interfaces. We can install a bundle from a URL. If you are using Knopflerfish, we may also open a file selector and browse the local file system.
- Using a bundle repository: The Oscar Bundle Repository (OBR) bundle enriches the user interface by providing a feature that allows selecting bundles from a "bundle repository" and automatically finding and installing all bundles that provide the packages needed by a given bundle. If we want to install our own bundles that way, we have to build our own repository, as described in the next section.

#### 2.3.2.5 How to create an OBR repository

A bundle repository is an XML file that describes a set of OSGi bundles. The bundle repository features and the descriptor syntax are described at [obr]. There are multiple ways to access the repository bundles:

***Web access*** – by associating an appropriate XSL file with the XML repository descriptor, all bundles in the OBR repository are accessible via a web page.

***Programmatic access*** – the Bundle Repository bundle provides an OSGi service for dynamically deploying repository bundles and the transitive closure of their deployment dependencies into an executing OSGi framework.

***Interactive access*** – the Bundle Repository bundle also provides a shell command for Oscar's Shell Service bundle, so if we are using Oscar we can interactively type OBR commands at Oscar's shell prompt. Additionally, the Shell Plugin bundle provides a GUI interface for the Shell GUI bundle.

The sample project contains a tool that allows a repository XML file to be generated automatically from a directory containing bundles. This tool browses the contents of the directory, extracts information from the manifests and builds the repository.xml file accordingly. The directory must be structured in a given manner: it may either contain directly the bundles, or a list of subdirectories with a naming convention – the directory <bundle> must contain a bundle named <bundle>.jar, and optionally the documentation of this bundle in a "doc" subdirectory, and the sources in a file named <bundle>-src.jar. The doc and sources are useful when viewing the repository.xml file in a browser.

Once our repository.xml file has been created and put at some accessible URL, it can be used by the OBR bundle. Information about how to configure Oscar to use a new repository descriptor is found at [obr]. It may also be viewed in a browser, provided a convenient XSL file is provided. An XSL file is provided with the tool; we may edit it to customize the title and description of our repository.

#### 2.3.2.6 How to add bundles to the Amigo bundle repository

Once our set of bundles is in a stable state, we may want to add it to the main Amigo bundle repository. This will be done by referring to our own repository.xml file from the main Amigo repository file, visit http://amigo.gforge.inria.fr/obr/v2/repository.xml .

Currently the main Amigo repository refers to the CMS and the ANS bundle repositories.

### 2.3.3 Some Amigo services using the OSGi framework

Context Management Service (CMS): a repository of OSGi bundles linked to context management (context source broker; helper API to write Amigo context sources and sinks) is available at http://core.lab.telin.nl/ãmigo/obr/repository.xml.

Awareness Notification Service (ANS): a repository of OSGi bundles linked to ANS is available at http://amigo.gforge.inria.fr/obr/ans/repository.xml.

Palantir: a repository of bundles corresponding to the WP7 Palantir demonstrator is available at http://amigo.gforge.inria.fr/obr.palantir/repository/repository.xml.

### 2.3.4 Resources

[amigo_obr] http://amigo.gforge.inria.fr/obr/v2/repository.xml. Contains the core Amigo bundles, some examples (referred to in the user's guide), a copy of the standard Oscar bundle repository, and references to the CMS OSGi repository.

[amigo_dev] http://amigo.gforge.inria.fr/obr/tutorialdeveloper_guide/. Amigo OSGI framework developer's guide.

[amigo_user] http://amigo.gforge.inria.fr/obr/tutorial/user_guide/. Amigo OSGI framework user's guide.

[Knopflerfish] http://www.knopflerfish.org/

[obr] http://oscar-osgi.sourceforge.net/repo/bundlerepository/. Documentation of the Oscar Bundle Repository bundle.

[oscar] http://oscar.objectweb.org/. The detailed Java documentation of each module is available from the OSGi Amigo repository [amigo_obr], except for the log4j bundle which is a simple encapsulation of existing Open Source libraries. The log4j documentation can be found at http://logging.apache.org/log4j/docs/.

[amigo_sample] http://amigo.gforge.inria.fr/obr/tutorial/full_tutorial_v2.zip. A sample project based on the OSGi framework.

# 3 How to develop a secure service

## 3.1 Overview

### 3.1.1 Objectives

The Amigo security framework is responsible for securing access to the services in the home. The Amigo security model is based on the Kerberos protocol, with shared secrets to establish mutual authentication. The security service is the mutually trusted Kerberos authority and acts as a trust broker between users and services inside the home and grants or denies access to services. The employed access control scheme in Amigo is role-based with few, well-known user, device, and service roles, thus enabling non-technical users to setup a secure system easily. As usual with Kerberos, access is granted or denied centrally by the security service and is relayed to users and services with encrypted tickets.

### 3.1.2 Principles

Accordingly, the implementation of Amigo security is split into two major components: the security service, which needs to be deployed in the home to enable secure use of services, and the security API that enables secure communication with the security service and between clients and services in the home. The security service as such is, from the service developer's point of view, a fixed service that is only communicated with via the security API. Thus, the security service is irrelevant for the purposes of this document.

The security API is available both as a Java OSGi bundle and as a .Net library, and uses lower level middleware functionality like WS-Discovery.

Before we go into more details though, it will help to quickly recapitulate the principles of Kerberos [Kerberos]. In Kerberos, security and privacy are achieved via a mutually trusted authority that shares a secret – a key for a symmetric cipher like AES, Triple DES, or Blowfish – with each of the participants of the security scheme. Using some coarse simplification[3], a typical exchange that enables secure and authorized of a client to a service has the following steps:

1. The client C requests a ticket granting ticket from the authority A; this request is sent unencrypted.
2. If the client is known, then A sends a ticket granting ticket TGT, which is encrypted with the secret $K_C$, which is known only to C and A.
3. C decrypts the TGT with $K_C$, which proves that it really came from A (because only A and C know $K_C$).
4. C uses information from TGT to create a ticket request for the service S that it wants to use and sends it to A.
5. A checks that the ticket request has been sent from C. It determines if A has permission to use S.
6. A sends a ticket $T_S$ to C, which contains information that is encrypted with $K_S$, the key that is shared between A and S.
7. C uses $T_S$ to prove to S that is has A's authorization to access S.
8. S attempts to decrypt $T_S$ with $K_S$, and if it does so successfully, grants access to C.

[3] This simplification is here only for didactic purposes; the implementation does not use this simplification.

In addition to this protocol, we need a way to initially establish the keys that are shared between the C (or S) and A. This is part of the registration procedure, which is covered in the subsequent sections.

In the remainder of this chapter, we will discuss how to use the .Net version of the security API to implement and consume secure services. The Java implementation uses the same object model so that the principles are sufficiently similar to be transferred from .Net to Java.

### 3.1.3 Assessment

The Amigo security libraries enable authenticated and authorized service use in the Amigo home, based on a simple, role-based, security concept. Using the API presented in this chapter, use of these mechanisms is significantly simplified, because the security protocol is abstracted into a comprehensive API that supports easy service and client implementation.

## 3.2 How to develop a secure service

In the following, we will describe the necessary steps to develop a secure Amigo service. We do this by walking through the code sample that is installed as part of installing the Amigo security package.

### 3.2.1 Service implementation

The following example shows the code that is necessary to implement a web service with secured methods. Hosting, registration, and persisting are handled in the next section. Detailed API documentation is available from the help files installed with the security package.

```
// ==++==
//
//
//   Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
//
//   The use and distribution terms for this software are contained in the
//   file named license.txt, which can be found in the root of this
//   distribution.
//   By using this software in any fashion, you are agreeing to be bound by
//   the terms of this license.
//
//   You must not remove this notice, or any other, from this software.
//
//
// ==--==

using System;
using System.Web.Services;
using EMIC.HomeFx.Security;
using EMIC.WSDiscovery.Server;
```

These declarations import the necessary namespaces – note that we are also importing the EMIC.WSDiscovery.Server namespace, which contains the .Net implementation of WS-Discovery.

```
namespace SecuredSeviceSample {
   [Scope ("urn:amigo-security-sample")]
   [Type ("sample-service", "urn:amigo-security-sample")]
   public class Service : EMIC.HomeFx.Security.Service {
```

The service implementations inherits from EMIC.HomeFx.Security.Service, which contains the necessary primitives to validate incoming requests. EMIC.HomeFx.Security.Service in turn inherits from EMIC.WSDiscovery.Server.Discoverable service, which enables automatic

discovery of our service. The Type and Scope attribute specify the WS-Discovery type and scope for this service.

```
public Service(EMIC.HomeFx.Security.Service securedService)
   : base(securedService){}
```

The only means of creating an EMIC.HomeFx.Security.Service is by copying from another instance of that class. Such an instance can be created by either registering a service with the security authority or by deserializing a persisted instance. Both possibilities are discussed in the next section.

```
[WebMethod]
public SecuredResult<string> Hello(SecuredRequest<string> request) {
```

A secure service has one of several web methods, which will be reachable once the service is hosted by the web server. In the current version, security has to be added explicitly to the service behaviour as shown in the example above. For this, the web method accepts one generic argument of type SecuredRequest<T0, T2, …, Tn>. The types T0,…,Tn are instantiated with the actual types of the unsecured version of the web method. Currently, up to five arguments are supported. Return type of the function is SecuredResult<T>, where T is the return type of the unsecured function.

```
      return ValidateAndDispatchRequest<string, string>(request, Hello);
   }
```

Inside the secured function, the request is dispatched to the unsecured, private version of the secured function via the ValidateAndDispatchRequest<T, T0, …, Tn> member function of EMIC.HomeFx.Security.Service. Based on the result of the validation of the incoming secured request, this function will either decrypt the encrypted parameters, pass them to the unsecured function, and return the encrypted result, or – in case of an unsuccessful validation – will return an error result. In either case, the return value of this dispatched function can directly be passed to the invoker of the secure web method.

```
   private string Hello(string name) {
      return String.Format("Hello {0} ({1}@{2})!",
         name,
         InvocationContext.User == null ? "<unknown>" :
         InvocationContext.User.ToString(),
         InvocationContext.Device);
   }
  }
}
```

The unsecured version of the function can assume that the caller has been successfully authenticated and authorized to use the service. If necessary, access to the authentication information, i.e., identity of the calling user and device, are available via the InvocationContex.User and InvocationContext.Device properties.

This concludes the discussion of the service code.

### 3.2.2 Registering and hosting a secured service

In the following, we discuss a small program that is capable of registering the secured service from the previous section with the security service (or retrieve a previous registration if it exists) and host the services with the Amigo .Net web server component.

```
// ==++==
//
//
//   Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
//
```

```
//    The use and distribution terms for this software are contained in the
//    file named license.txt, which can be found in the root of this
//    distribution.
//    By using this software in any fashion, you are agreeing to be bound by
//    the terms of this license.
//
//    You must not remove this notice, or any other, from this software.
//
//
// ==--==

using System;
using System.Xml.Serialization;
using System.Xml;
using System.Net;

using EMIC.WebServer;
using EMIC.HomeFx.Security;
using EMIC.HomeFx.Security.Proxies;
using EMIC.WSAddressing;
```

Import of the required namespaces for hosting and communication with the security service (in EMIC.HomeFx.Security.Proxies).

```
namespace SecuredSeviceSample {

   class Program {
      static readonly string registrationFileLocation =
                        System.Windows.Forms.Application.LocalUserAppDataPath
                        + @"\registration.xml";
```

We store registration information in the user path for this application. Of course, other choices are possible. Care should be taken to secure the stored information because it enables an attacker to impersonate the service.

```
      static int Main(string[] args)
      {
         EMIC.HomeFx.Security.Service securedService = null;
         XmlSerializer xmlSerializer =
           new XmlSerializer(typeof(EMIC.HomeFx.Security.Service));
         try
         {
            using (XmlReader xmlReader = XmlReader.Create(registrationFileLocation))
            {
               securedService =
                 (EMIC.HomeFx.Security.Service)xmlSerializer.Deserialize(xmlReader);
               securedService.SecurityServiceProxy =
                 RediscoveringSecurityServiceProxy.Instance;
            }
         }
```

If possible, we try to recreate an EMIC.HomeFx.Security.Service.Service from information stored at the designated location. If this succeeds, we set the SecurityServiceProxy field of the service to the singleton instance provided by the RediscoveringSecurityServiceProxy, which supports automatic finding of the security service and automatic fail-over to alternative security services in case of communication failures. In case that an exception is thrown during the restoration process, we (re-)register the service with the security service:

```
         catch
         {
            string password = PronounceablePasswordGenerator.GeneratePassword(8);
            Console.WriteLine("Attempting to register with password " + password);

            try
            {
               securedService = EMIC.HomeFx.Security.Service.RegisterNewService(
                  "Amigo Security Sample Service",
                  System.Net.Dns.GetHostName(),
                  password,
                  ServiceRole.Unrestricted,
                  RediscoveringSecurityServiceProxy.Instance);
```

```
        }
```

For this, we create a new password that is used to encrypt the registration process (and will need to be transferred out of band to the security service). Then, we call RegisterNewService with the registration information of this service. We specify its requested service role (which controls client authorization for this service at the security service) and again provide the singleton instance provided by the RediscoveringSecurityServiceProxy to enable communication with the security service.

```
        catch (RediscoveryException) {
            Console.WriteLine("Could not locate security service");
            return -1;
        }
        catch (RegistrationException e) {
            Console.WriteLine("Could not register with security service: " +
                e.Message);
            return -1;
        }
```

During this process, irrecoverable communication errors will lead to a RediscoveryException. A registration attempt that is rejected from the security service for some reason will lead to a Registration exception. In either case, the service has not been successfully registered and we terminate the program with an error code.

```
        try {
            using (XmlWriter xmlWriter = XmlWriter.Create(registrationFileLocation))
            {
                xmlSerializer.Serialize(xmlWriter, securedService);
            }
        }
        catch {
            Console.WriteLine("Warning: Could not persist registration information");
        }
    }
```

Upon successful registration, we attempt to persist the registration information to the designated location.

If the program has successfully executed up to here, then we have created an instance of EMIC.HomeFx.Security.Service.Service, which we can use to create and host an instance of the secured service from the previous section:

```
        WebServer.Port = 0;
        WebServer.Start();

        using (Service service = new Service(securedService)) {
            WebServer.AddWebService(serviceLocation, service);
            Console.WriteLine("Sample service running." + Environment.NewLine
              + "Press <return> to quit");
            Console.ReadLine();
        }
        return 0;
    }
  }
}
```

This concludes our explanation of how to develop secured Amigo services.

## 3.3 How to develop a secure service client

In a similar manner, we will now explore the steps necessary to access a secured Amigo client:

```
// ==++==
//
//
//    Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
```

```
//
//   The use and distribution terms for this software are contained in the file
//   named license.txt, which can be found in the root of this distribution.
//   By using this software in any fashion, you are agreeing to be bound by the
//   terms of this license.
//
//   You must not remove this notice, or any other, from this software.
//
//
// ==--==

using System;
using System.Net;
using System.Threading;

using EMIC.HomeFx.Security;
using EMIC.HomeFx.Security.Proxies;
using EMIC.WSDiscovery;
using EMIC.WSDiscovery.Client;
```

Import of the necessary namespaces for Amigo security.

```
namespace SecuredClientSample {

   class Program {
      static int Main(string[] args)
      {
         if (!Device.IsRegistered())
         {
            Console.Error.WriteLine("The device has not been registered.");
            return -1;
         }
```

Instead of a explicit registration step, we assume that registration of the device and user of this client program has been performed separately. The Amigo security package contains an application that serves exactly that purpose. If the device has not previously been registered, access to secured Amigo services is not possible and we exit with an error code.

```
         Device thisDevice = Device.ThisDevice(RediscoveringSecurityServiceProxy.Instance);
         Authentication authentication = thisDevice.AuthenticateDevice();
```

In the next step, we create an object represention of this device and authenticate it with the security service that is reachable via the singleton instance of RediscoveringSecurityServiceProxy. This will obtain the ticket granting ticket from the security service and store it locally on the device. If other clients on this have recently obtained such a ticket, it will be reused until it expires.

```
         using (ProbeClient probeClient = new ProbeClient())
         {
            probeClient.AddServiceScope(
              new ServiceScope(new Uri("urn:amigo-security-sample")));
            probeClient.AddServiceType(
              new ServiceType("sample-service", "urn:amigo-security-sample"));

            while (probeClient.ProbeMatches == null || probeClient.ProbeMatches.Count == 0)
            {
               Console.WriteLine("Probing for service...");
               probeClient.SendProbe();
               Thread.Sleep(2000);
            }
```

We probe for the service that we want to use until we find one, using the WS-Discovery implementation.

```
            ServiceInfo match = probeClient.ProbeMatches[0];
            Sid sid = Sid.GetSidFromServiceInfo(match);
```

We retrieve the unique name under which this service is known to the security service. This name is included into the metadata transmitted in the ProbeMatch message from the service.

```
EMIC.HomeFx.Security.Authorization authorization
  = thisDevice.Authorize(sid, authentication, new TimeSpan(1, 0, 0));

SessionCredentials credentials = authorization.NewSessionCredentials();
```

We try to obtain a ticket for the service, with a validity period of at least one hour. Then we use this ticket to create the necessary credentials to start a new session with the service.

```
using (ServiceProxy proxy = new ServiceProxy())
{
   proxy.Proxy = new WebProxy();
   proxy.Url = match.FirstXaddr.AbsoluteUri;
```

We prepare a proxy to contact the service, using the address from its ProbeMatch reply.

```
   string response = thisDevice.InvokeSecuredService<string, string>(
      credentials,
      proxy.Hello,
      "Amigo Security Client");
```

Finally, we use the InvokeSecuredService<T, T0,…Tn> method of the Device class to issue a secured call to the service. As parameters, this method takes the credentials that have been created in the previous step, a delegate to the method that is responsible to send the secured request to the service, and the service call's arguments of type T0, …, Tn.

```
      Console.WriteLine("Response = \"" + response + "\"");
      Console.ReadLine();
   }
  }
  return 0;
 }
}
}
```

## 3.4 Resources

[Kerberos] John Kohl and B. Clifford Neuman. The Kerberos Network Authentication Service (Version 5). Internet Request for Comments RFC-1510. September 1993.

# 4 How to develop a semantic service

## 4.1 Overview

This chapter presents a 'how-to' for the Amigo service developer that describes how to create semantic service descriptions for Amigo services. Any basic Amigo service (see Chapter 2), e.g., a domotic service (see Chapter 6) or other, can be enriched with a semantic description. This enables much more flexibility and effectiveness when discovering a service or dynamically integrating it in a service-based application (see Chapter 5). Semantic service descriptions are created using the Amigo-S language, which is presented in Chapter 4 of [D3.2]. In addition to presenting the principles, features, and an initial assessment of Amigo-S service description, this chapter aims to provide several concrete examples of using Amigo-S to describe services. These include examples of using the Amigo-S ontology and vocabularies, of creating functional descriptions for a service, of describing non-functional properties of the service including context-aware and quality of service properties, as well as how to register these descriptions with the semantic service repository.

### 4.1.1 Objectives and principles

The Amigo-S description language is a declarative language for the semantic specification of Amigo services. Amigo-S uses OWL-S as a basis. The Ontology Web Language (OWL) is a recommendation by W3C supporting formal description of ontologies and reasoning on them. An ontology can represent concepts of any knowledge domain and relations between them. OWL-Services (OWL-S) is an OWL-based ontology for semantically specifying Web services [OWL-S].

However, OWL-S cannot be used as-is for describing Amigo-aware services for several reasons. Firstly, the only concrete grounding with an interaction protocol that is defined in OWL-S is the mapping of OWL-S processes to WSDL operations. Indeed, OWL-S has been defined for semantically describing Web services. In the Amigo home, Web services will be used together with other technologies, and we need a semantic description language that could be used for all of them, independently of the underlying technology. Thus, OWL-S is extended by enabling several groundings to be employed for a service.

Secondly, OWL-S lacks support for describing context-ware and quality of service related information, which are key non-functional properties that we want to describe for Amigo-aware services. Therefore, included in the Amigo-S language are generic classes for describing such non-functional properties.

In addition, it is desired for Amigo-S to have the possibility of specifying these properties globally for all the functionalities that an Amigo service provides, as well as individually for each functionality. Thus, OWL-S is extended such that these properties could be expressed at different levels.

Thirdly, we wish to be able to semantically describe the event capabilities an Amigo service has, and Amigo-S provides extensions to allow this. Please refer to [D2.1] for a more detailed analysis of OWL and OWL-S with respect to Amigo, and to [D3.2] for further detail on novel aspects of Amigo-S.

Amigo-S reuses classes that are already formally specified as part of OWL-S, aiming to be consistent with similar existing concepts, and thus reduce the effort for learning a new language for developers who are already familiar with writing OWL-S descriptions.

The semantic concepts and vocabulary used in Amigo-S service descriptions are provided by a common set of vocabulary ontologies. These vocabularies are modularly-defined to support maintainability and future evolution of concepts related to the Amigo home. The vocabularies include the Amigo Core Concepts which define the basic vocabulary that helps to tie the other

vocabularies together, Devices and Platforms that provides a classification of different platforms hosted by devices and a generic classification of device types and their states, Functional Capabilities which includes several concepts representing the function of an individual capability, Quality of Service which defines a range of QoS parameters, User Context and Physical Context which attempts to model all context parameters that may potentially be related to the Amigo user and the generic parameters that are related to the elements of the physical environment respectively, and the Multimedia vocabulary that describes the different types of content that can be processed by the devices in an Amigo home. There are also several other domain-specific vocabularies defined.

By incorporating shared, semantic concepts from common vocabulary ontologies in Amigo-S service descriptions, we can promote a high level of interoperability between different Amigo services and applications. Please refer to [D3.2] for further information on the available vocabularies.

In order to support the use of the full range of features of Amigo-S for service registration and discovery, and again to increase the availability and promote the interoperability of Amigo services, a common, dedicated semantic service repository is provided. The use of the repository is covered later in this chapter.

### 4.1.2 Features

The main features of the Amigo-S service description are that:

- It allows semantic service descriptions to be created for basic services, where the service provides a collection of atomic capabilities.
- Semantic service descriptions can provide semantic concepts for a whole service, for each capability and its inputs and outputs, and optionally its pre-conditions and effects.
- It allows semantic descriptions of conversation-based services, where the capabilities the service provides are described as a workflow of sub-capabilities.
- Complex workflows can be created by composing capabilities using a variety of control constructs, e.g. Sequence, Choice.
- Context parameters can be described for a service, both at the level of individual capabilities, and for the service as a whole.
- It supports and open-ended number of context parameters, based on the available context sources.
- The quality of service parameters a service supports can be described both at the individual capability level, and for the service as a whole.
- It supports a range of commonly required quality of service parameters, including: throughput, cost, latency, response time, error rate, jitter, data encryption, and accessibility.
- It allows semantic descriptions of a service's event capabilities, specifically its event production and consumption.
- The descriptions of a service's event capabilities may define several profiles (event or command). As a result, an event-based service can receive and send multiple events and commands.
- Multiple groundings can be supported.

In addition, a dedicated semantic service repository for services described by Amigo-S is provided. The repository itself is implemented as a basic Amigo service employing the Amigo OSGi programming and deployment framework (see Chapter 2). The repository can perform

exact and weak semantic service matching at the service and capability levels, as well as straight service retrieval.

The semantic service repository can also offer support for legacy (UPnP) services through the use of the Amigo interoperability framework. Please refer to [D3.1b] and [D3.2] for details.

### 4.1.3 Assessment

The advantages of the Amigo-S service description include:

- By creating a semantic service description for a basic service, it allows the service to be discovered via semantic matching at both the service and capability levels, increasing the service's availability and promoting its interoperability.
- Describing a semantic service's provided capabilities as conversation-based workflows allows the expression of data and control dependencies between the service's capabilities.
- By allowing a service's description to contain context parameters, the service can be incorporated by the context-aware discovery mechanisms (please refer to Chapter 5 for details), which enables an application to optimise service selection based on services' current context, and frees the application developer from the need to actively search for the optimal service.
- By allowing a service's description to feature quality of service parameters, it makes the service available to applications which employ the priority-based quality of service selection model provided by the Quality of Service Aware Service Selection Tool (QASST), which allows the Amigo middleware to serve as many requests as possible while satisfying the majority of Amigo users.
- The event-based service description features allows programmers to focus on the event-based service interface in terms of typed high-level operations, namely command and event. Providing a high-level viewpoint prevents programmers from dealing with low-level operations of heterogeneous communication modes when it is unnecessary.
- Enabling support for multiple groundings promotes interoperability between different service technologies.

As all Amigo-S services can be registered with the common semantic service repository, the Amigo-S service developer is presented with a common feel and interface regardless of what style, or mix of styles, of service they develop. Furthermore, the use of the common semantic service repository facilitates interoperability between the different styles of service. For example, an application can employ a service that features context aware parameters, whether the application uses the context-aware features or not.

## 4.2 How to write a semantic service description

### 4.2.1 Functional service description

In this section, we shall look at how to create semantic services in Amigo-S. Two simple examples are provided – the first is a multimedia browser service that allows a user to explore the contents of a multimedia library, and the second is a multimedia renderer service that permits different media to be played.

For each example, we shall look at how the service can be created based on the Amigo-S language using the Protégé ontology editor, and also provide the raw OWL XML that makes up the service description.

The Amigo-S developer has several tools at his disposal to assist him in creating service descriptions. The Protégé ontology editor can be used to graphically edit descriptions [Protégé]. The Eyeball OWL checker tool, available from the Jena project website, is a lint-like command-line tool that can be used to check that service descriptions are free of certain common OWL errors [Eyeball]. Developers who prefer to work directly with Amigo-S XML can of course create service descriptions using their favourite text editor.

For clarity, examples present the full OWL XML of the service description both in this chapter, and later in Chapter 5. The examples can be cut and paste into OWL files using the developer's tools of choice, and may also act as templates for developing new descriptions.

We will first look at the multimedia browser service. This service has two provided capabilities – a ListMedia capability that provides a list of all of the media in the library, and a ListMostPlayedMedia capability that provides a list of the twenty-five most played media for a particular user. In order for the ListMostPlayedMedia capability to identify the user, the user must first login before the most played list is created. As such, the ListMostPlayedMedia capability is conversation-based, and comprises a sequential workflow of a Login sub-capability followed by a CreateMostPlayedMediaList sub-capability. By describing this capability as workflow, it ensures that these sub-capabilities are carried out in this order, and that both are performed by the same service.

We will now begin to create this service. First, we must include all the necessary pre-amble:

```
<?xml version="1.0"?>
<!DOCTYPE uridef[
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
  <!ENTITY objList "http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl">
  <!ENTITY wsdl "http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.wsdl">
]>
<rdf:RDF
  xmlns:lang="http://www.hitech-projects.com/euprojects/amigo/Amigo-S.owl#"
  xmlns:capabilities="http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#"
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:objList="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  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:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns="http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.owl#"
  xmlns:wsdl="http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.wsdl"
  xml:base="http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.owl">
<owl:Ontology rdf:about="">
 <owl:imports rdf:resource="http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl"/>
 <owl:imports rdf:resource="http://www.hitech-projects.com/euprojects/amigo/Amigo-S.owl"/>
</owl:Ontology>
```

This defines the XML namespaces for the Amigo-S language and capabilities, OWL-S, the multimedia browser description itself, and a WSDL description of the service implementation. Also, the base is set to the URI of the service description. Finally, the Amigo-S and capabilities documents are imported.

Next, we define the service profile for the multimedia browser service. The profile describes the capabilities a service has, as well as a textual name and description of the service, and a link to the grounding the service supports.

At this point, the OWL XML of the service profile looks like this:

```
<service:Service rdf:ID="MultimediaBrowserService">
   <service:presents>
    <capabilities:ServiceProfile rdf:ID="MultimediaBrowserProfile">
```

```
    <!-- Provided capabilities -->
    <lang:hasProvidedCapability>
      <capabilities:ListMedia rdf:ID="ListMediaCapability">
        <lang:hasConversation rdf:resource="#ListMediaConversation"/>
        <lang:hasOutput rdf:resource="#ListMediaOutput"/>
      </capabilities:ListMedia>
    </lang:hasProvidedCapability>

    <lang:hasProvidedCapability>
      <capabilities:ListMostPlayedMedia rdf:ID="ListMostPlayedMediaCapability">
        <lang:hasConversation rdf:resource="#ListMostPlayedMediaConversation"/>
        <lang:hasInput rdf:resource="#ListMostPlayedMediaInput"/>
        <lang:hasOutput rdf:resource="#ListMostPlayedMediaOutput"/>
      </capabilities:ListMostPlayedMedia>
    </lang:hasProvidedCapability>

    <!-- Sub-capabilities -->
    <lang:hasCapability>
      <capabilities:Login rdf:ID="LoginCapability">
        <lang:hasConversation rdf:resource="#LoginConversation"/>
        <lang:hasInput rdf:resource="#LoginInput"/>
        <lang:hasOutput rdf:resource="#LoginOutput"/>
      </capabilities:Login>
    </lang:hasCapability>

    <lang:hasCapability>
      <capabilities:CreateMostPlayedMediaList rdf:ID="CreateMostPlayedMediaListCapability">
        <lang:hasConversation rdf:resource="#CreateMostPlayedMediaListConversation"/>
        <lang:hasOutput rdf:resource="#CreateMostPlayedMediaListOutput"/>
      </capabilities:CreateMostPlayedMediaList>
    </lang:hasCapability>

    <service:presentedBy rdf:resource="#MultimediaBrowserService"/>

    <profile:textDescription rdf:datatype=http://www.w3.org/2001/XMLSchema#string">
      Example multimedia browser service.
    </profile:textDescription>

    <profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      MultimediaBrowser Service
    </profile:serviceName>
  </capabilities:ServiceProfile>
 </service:presents>

 <service:supports rdf:resource="#MultimediaBrowserServiceGrounding"/>
</service:Service>
```

Each capability description states the semantic that represents the capability, e.g. ListMedia and ListMostPlayedMedia, a link to its conversation description, and links to descriptions of any inputs and outputs it has.

ListMedia is implemented with an atomic conversation, and has a single output. The OWL XML for the ListMedia conversation and output description looks like:

```
<!-- Conversations -->
<process:AtomicProcess rdf:ID="ListMediaConversation">
 <process:hasOutput>
  <process:Output rdf:ID="ListMediaOutput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#MediaList
   </process:parameterType>
  </process:Output>
 </process:hasOutput>
</process:AtomicProcess>
```

ListMostPlayedMedia is implemented as a composite conversation, comprised of the sequence of the Login and CreateMostPlayedMediaList sub-capabilities. Each of these sub-capabilities is implemented as an atomic conversation. The OWL XML for conversation for the ListMostPlayedMedia provided capability looks like:

```
<process:CompositeProcess rdf:ID="ListMostPlayedMediaConversation">
```

```
<process:hasInput>
 <process:Input rdf:ID="ListMostPlayedMediaInput">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
   http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#User
  </process:parameterType>
 </process:Input>
</process:hasInput>

<process:hasOutput>
 <process:Output rdf:ID="ListMostPlayedMediaOutput">
  <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
   http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#MediaList
  </process:parameterType>
 </process:Output>
</process:hasOutput>

<process:composedOf>
 <process:Sequence>
  <process:components>
   <process:ControlConstructList>
    <objList:first>
     <process:Perform rdf:ID="LoginPerform">
      <process:process rdf:resource="#LoginCapability"/>
      <!-- Parameter Bindings -->
     </process:Perform>
    </objList:first>
    <objList:rest>
     <process:ControlConstructList>
      <objList:first>
       <process:Perform rdf:ID="CreateMostPlayedMediaListPerform">
        <process:process rdf:resource=
         "#CreateMostPlayedMediaListCapability"/>
        <!-- Parameter Bindings -->
       </process:Perform>
      </objList:first>
      <objList:rest rdf:resource="&objList;#nil"/>
     </process:ControlConstructList>
    </objList:rest>
   </process:ControlConstructList>
  </process:components>
 </process:Sequence>
</process:composedOf>
</process:CompositeProcess>
```

and the Login and CreateMostPlayedMediaList sub-capabilities look like:

```
<process:AtomicProcess rdf:ID="LoginConversation">
 <process:hasInput>
  <process:Input rdf:ID="LoginInput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#User
   </process:parameterType>
  </process:Input>
 </process:hasInput>
</process:AtomicProcess>

<process:AtomicProcess rdf:ID="CreateMostPlayedMediaListConversation">
 <process:hasOutput>
  <process:Output rdf:ID="CreateMostPlayedMediaListOutput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#MediaList
   </process:parameterType>
  </process:Output>
 </process:hasOutput>
</process:AtomicProcess>
```

At this point, the description of the structure of the multimedia browser service is complete. What remains is to link the atomic conversation with their representative operation in the service's grounding. The grounding makes references to the service's WSDL file, which is provided when the service is registered with the repository. The OWL XML for the multimedia browser service grounding looks like:

```
<!-- Grounding -->
<grounding:WsdlGrounding rdf:ID="MultimediaBrowserServiceGrounding">
```

```
  <grounding:hasAtomicProcessGrounding rdf:resource="#ListMediaGrounding"/>
  <grounding:hasAtomicProcessGrounding rdf:resource="#LoginGrounding"/>
  <grounding:hasAtomicProcessGrounding rdf:resource="CreateMostPlayedMediaListGrounding"/>
 </grounding:WsdlGrounding>

 <grounding:WsdlAtomicProcessGrounding rdf:ID="ListMediaGrounding">
  <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
   http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.wsdl
  </grounding:wsdlDocument>

  <grounding:owlsProcess rdf:resource="#ListMediaConversation"/>

  <grounding:wsdlOperation>
   <grounding:WsdlOperationRef>
    <grounding:portType rdf:datatype="&xsd;#anyURI">
     &wsdl;#MultimediaBrowser
    </grounding:portType>
    <grounding:operation rdf:datatype="&xsd;#anyURI">
     &wsdl;#listMedia
    </grounding:operation>
   </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>

  <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
   &wsdl;#listMediaRequest
  </grounding:wsdlInputMessage>

  <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
   &wsdl;#listMediaResponse
  </grounding:wsdlOutputMessage>

  <grounding:wsdlOutput>
   <grounding:WsdlOutputMessageMap>
    <grounding:owlsParameter rdf:resource="#ListMediaOutput"/>
    <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
     &wsdl;#listMediaReturn
    </grounding:wsdlMessagePart>
   </grounding:WsdlOutputMessageMap>
  </grounding:wsdlOutput>
 </grounding:WsdlAtomicProcessGrounding>

 <grounding:WsdlAtomicProcessGrounding rdf:ID="LoginGrounding">
  <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
   http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.wsdl
  </grounding:wsdlDocument>

  <grounding:owlsProcess rdf:resource="#LoginConversation"/>

  <grounding:wsdlOperation>
   <grounding:WsdlOperationRef>
    <grounding:portType rdf:datatype="&xsd;#anyURI">
     &wsdl;#MultimediaBrowser
    </grounding:portType>
    <grounding:operation rdf:datatype="&xsd;#anyURI">
     &wsdl;#login
    </grounding:operation>
   </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>

  <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
   &wsdl;#loginRequest
  </grounding:wsdlInputMessage>

  <grounding:wsdlInput>
   <grounding:WsdlInputMessageMap>
    <grounding:owlsParameter rdf:resource="#LoginInput"/>
    <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
     &wsdl;#user
    </grounding:wsdlMessagePart>
   </grounding:WsdlInputMessageMap>
  </grounding:wsdlInput>

  <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
   &wsdl;#loginResponse
  </grounding:wsdlOutputMessage>
```

```
  </grounding:WsdlAtomicProcessGrounding>

  <grounding:WsdlAtomicProcessGrounding rdf:ID="CreateMostPlayedMediaListGrounding">
   <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/MultimediaBrowser.wsdl
   </grounding:wsdlDocument>

   <grounding:owlsProcess rdf:resource="#CreateMostPlayedMediaListConversation"/>

   <grounding:wsdlOperation>
    <grounding:WsdlOperationRef>
     <grounding:portType rdf:datatype="&xsd;#anyURI">
      &wsdl;#MultimediaBrowser
     </grounding:portType>
     <grounding:operation rdf:datatype="&xsd;#anyURI">
      &wsdl;#createMostPlayedMediaList
     </grounding:operation>
    </grounding:WsdlOperationRef>
   </grounding:wsdlOperation>

   <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
    &wsdl;#createMostPlayedMediaListRequest
   </grounding:wsdlInputMessage>

   <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
    &wsdl;#createMostPlayedMediaListResponse
   </grounding:wsdlOutputMessage>

   <grounding:wsdlOutput>
    <grounding:WsdlOutputMessageMap>
     <grounding:owlsParameter rdf:resource="#CreateMostPlayedMediaListOutput"/>
     <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
      &wsdl;#createMostPlayedMediaListReturn
     </grounding:wsdlMessagePart>
    </grounding:WsdlOutputMessageMap>
   </grounding:wsdlOutput>
  </grounding:WsdlAtomicProcessGrounding>
```

Next, the multimedia renderer service is presented. For simplicity of the example, the service supports a single PlayMedia operation, which features a composite conversation comprising of the sequence of SetMedia and Play sub-capabilities. The service can be created using the developer's tools of choice, using the following OWL XML:

```
<?xml version="1.0"?>
<!DOCTYPE uridef[
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
  <!ENTITY objList "http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl">
  <!ENTITY wsdl
    "http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.wsdl">
]>
<rdf:RDF
  xmlns:lang="http://www.hitech-projects.com/euprojects/amigo/Amigo-S.owl#"
  xmlns:capabilities="http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#"
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:objList="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  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:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns="http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.owl#"
  xml:base="http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.owl"
  xmlns:wsdl="http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.wsdl">
<owl:Ontology rdf:about="">
 <owl:imports rdf:resource="http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl"/>
 <owl:imports rdf:resource="http://www.hitech-projects.com/euprojects/amigo/Amigo-S.owl"/>
</owl:Ontology>

  <service:Service rdf:ID="MultimediaRendererService">
```

```
  <service:presents>
   <capabilities:ServiceProfile rdf:ID="MultimediaRendererProfile">

     <!-- Provided capabilities -->
     <lang:hasProvidedCapability>
      <capabilities:PlayMedia rdf:ID="PlayMediaCapability">
       <lang:hasConversation rdf:resource="#PlayMediaConversation"/>
       <lang:hasInput rdf:resource="#PlayMediaInput"/>
       <lang:hasOutput rdf:resource="#PlayMediaOutput"/>
      </capabilities:PlayMedia>
     </lang:hasProvidedCapability>

     <!-- Sub-capabilities -->
     <lang:hasCapability>
      <capabilities:SetMedia rdf:ID="SetMediaCapability">
       <lang:hasConversation rdf:resource="#SetMediaConversation"/>
       <lang:hasInput rdf:resource="#SetMediaInput"/>
      </capabilities:SetMedia>
     </lang:hasCapability>

     <lang:hasCapability>
      <capabilities:Play rdf:ID="PlayCapability">
       <lang:hasConversation rdf:resource="#PlayConversation"/>
       <lang:hasOutput rdf:resource="#PlayOutput"/>
      </capabilities:Play>
     </lang:hasCapability>

     <service:presentedBy rdf:resource="#MultimediaRendererService"/>

     <profile:textDescription rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      Example multimedia renderder service.
     </profile:textDescription>

     <profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      MultimediaRenderer Service
     </profile:serviceName>
   </capabilities:ServiceProfile>
  </service:presents>

  <service:supports rdf:resource="#MultimediaRendererServiceGrounding"/>
 </service:Service>

 <!-- Conversations -->
 <process:CompositeProcess rdf:ID="PlayMediaConversation">
  <process:hasInput>
   <process:Input rdf:ID="PlayMediaInput">
    <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#Media
    </process:parameterType>
   </process:Input>
  </process:hasInput>

  <process:hasOutput>
   <process:Output rdf:ID="PlayMediaOutput">
    <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
     http://www.w3.org/2001/XMLSchema#boolean
    </process:parameterType>
   </process:Output>
  </process:hasOutput>

  <process:composedOf>
   <process:Sequence>
    <process:components>
     <process:ControlConstructList>
      <objList:first>
       <process:Perform rdf:ID="SetMediaPerform">
        <process:process rdf:resource="#SetMediaCapability"/>
        <!-- Parameter Bindings -->
       </process:Perform>
      </objList:first>
      <objList:rest>
       <process:ControlConstructList>
        <objList:first>
         <process:Perform rdf:ID="PlayPerform">
          <process:process rdf:resource="#PlayCapability"/>
```

```
            <!-- Parameter Bindings -->
          </process:Perform>
        </objList:first>
        <objList:rest rdf:resource="&objList;#nil"/>
      </process:ControlConstructList>
    </objList:rest>
  </process:ControlConstructList>
 </process:components>
</process:Sequence>
</process:composedOf>
</process:CompositeProcess>

<process:AtomicProcess rdf:ID="SetMediaConversation">
 <process:hasInput>
  <process:Input rdf:ID="SetMediaInput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#Media
   </process:parameterType>
  </process:Input>
 </process:hasInput>
</process:AtomicProcess>

<process:AtomicProcess rdf:ID="PlayConversation">
 <process:hasOutput>
  <process:Output rdf:ID="PlayOutput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.w3.org/2001/XMLSchema#boolean
   </process:parameterType>
  </process:Output>
 </process:hasOutput>
</process:AtomicProcess>

<!-- Grounding -->
<grounding:WsdlGrounding rdf:ID="MultimediaRendererServiceGrounding">
 <grounding:hasAtomicProcessGrounding rdf:resource="#SetMediaGrounding"/>
 <grounding:hasAtomicProcessGrounding rdf:resource="#PlayGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="SetMediaGrounding">
 <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
  http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.wsdl
 </grounding:wsdlDocument>

 <grounding:owlsProcess rdf:resource="#SetMediaConversation"/>

 <grounding:wsdlOperation>
  <grounding:WsdlOperationRef>
   <grounding:portType rdf:datatype="&xsd;#anyURI">
    &wsdl;#MultimediaRenderer
   </grounding:portType>
   <grounding:operation rdf:datatype="&xsd;#anyURI">
    &wsdl;#setMedia
   </grounding:operation>
  </grounding:WsdlOperationRef>
 </grounding:wsdlOperation>

 <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
  &wsdl;#setMediaRequest
 </grounding:wsdlInputMessage>

 <grounding:wsdlInput>
  <grounding:WsdlInputMessageMap>
   <grounding:owlsParameter rdf:resource="#SetMediaInput"/>
   <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
    &wsdl;#media
   </grounding:wsdlMessagePart>
  </grounding:WsdlInputMessageMap>
 </grounding:wsdlInput>

 <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
  &wsdl;#setMediaResponse
 </grounding:wsdlOutputMessage>
</grounding:WsdlAtomicProcessGrounding>

 <grounding:WsdlAtomicProcessGrounding rdf:ID="PlayGrounding">
```

```
  <grounding:wsdlDocument rdf:datatype="&xsd;#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.wsdl
  </grounding:wsdlDocument>

  <grounding:owlsProcess rdf:resource="#PlayConversation"/>

  <grounding:wsdlOperation>
    <grounding:WsdlOperationRef>
      <grounding:portType rdf:datatype="&xsd;#anyURI">
        &wsdl;#MultimediaRenderer
      </grounding:portType>
      <grounding:operation rdf:datatype="&xsd;#anyURI">
        &wsdl;#play
      </grounding:operation>
    </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>

  <grounding:wsdlInputMessage rdf:datatype="&xsd;#anyURI">
    &wsdl;#playRequest
  </grounding:wsdlInputMessage>

  <grounding:wsdlOutputMessage rdf:datatype="&xsd;#anyURI">
    &wsdl;#playResponse
  </grounding:wsdlOutputMessage>

  <grounding:wsdlOutput>
    <grounding:WsdlOutputMessageMap>
      <grounding:owlsParameter rdf:resource="#PlayOutput"/>
      <grounding:wsdlMessagePart rdf:datatype="&xsd;#anyURI">
        &wsdl;#playReturn
      </grounding:wsdlMessagePart>
    </grounding:WsdlOutputMessageMap>
  </grounding:wsdlOutput>
 </grounding:WsdlAtomicProcessGrounding>
</rdf:RDF>
```

This completes the creation of the semantic service descriptions for the multimedia browser and multimedia renderer service. Later in this chapter, we shall see how these services are registered with the runtime system, and later in Chapter 5, how they are used to compose a multimedia player service described by a user task.

### 4.2.2 Context-aware service description

To provide an example of how a context parameter is described, we shall consider an ambulance that has a certain location. This example is taken from a larger emergency response application scenario. Further details of this scenario are presented in Chapter 5. In this case, the service context parameter would describe its location in the longitude, latitude and altitude format using the following RDF fragment (namespaces are excluded for the sake of brevity.

```
<?xml version="1.0"?>
<rdf:RDF>
  <ServiceLocation>
    <probability>0.9</probability>
    <timestamp>2006-10-18T00:00:00</timestamp>
    <hasAbsoluteLocation>
      <AmigoICCS:WGS84Location rdf:ID="location1">
        <longitude>6.8897</longitude>
        <latitude>52.2328</latitude>
        <altitude>0.00</altitude>
      </AmigoICCS:WGS84Location>"
    </hasAbsoluteLocation>
  </ServiceLocation>
</rdf:RDF>
```

### 4.2.3 Quality of service aware service description

The QoS-aware Service Selection Tool (QASST) is a mechanism for filtering a list of services and selecting the most appropriate one which addresses specific QoS requirements set by an

Amigo User. In order for this to function properly, the semantic service description has to include the specification of some QoS features.

The example that will be used to demonstrate how a QoS-aware semantic service description should be provided is the common case of a video delivery service. Consider the case where two or more Media Servers maintain movies in the Amigo home environment and that the movie requested by the Amigo user is available in both. Then the QASST selects which of the two video services is the most appropriate for the user requesting the video delivery.

The following example illustrates such a video-on-demand service semantic description that includes several QoS properties (i.e. Throughput, Error Rate, Jitter, Latency, MTBF, MTTR, Availability, Accessibility, Response Time, Data Encryption, Supported Standard).

```
<?xml version="1.0" encoding="utf-8"?>
<Amigo:Service rdf:ID="VideoOnDemand@home1">
   <ServiceType rdf:datatype=
   "http://www.w3.org/2001/XMLSchema#string">VideoDelivery</ServiceType>
   <hasJitter>
      <Jitter rdf:ID="CD_QoSP_Jitter">
         <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">1</Amigo:QC_Value>
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">msec</Amigo:QC_Metric>
      </Jitter>
   </hasJitter>
   <hasLatency>
      <Latency rdf:ID="CD_QoSP_Latency">
        <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">300</Amigo:QC_Value>
        <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">msec</Amigo:QC_Metric>
      </Latency>
   </hasLatency>
   <hasAccessibility>
      <Accessibility rdf:ID="CD_QoSP_Accessibility">
         <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0.99000</Amigo:QC_Value>
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0</Amigo:QC_Metric>
      </Accessibility>
   </hasAccessibility>
   <hasResponseTime>
      <ResponseTime rdf:ID="CD_QoSP_ResponseTime">
         <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0.01</Amigo:QC_Value>
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">sec</Amigo:QC_Metric>
      </ResponseTime>
   </hasResponseTime>
   <hasMTBF>
      <MTBF rdf:ID="CD_QoSP_MTBF">
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">sec</Amigo:QC_Metric>
         <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">36000000</Amigo:QC_Value>
      </MTBF>
   </hasMTBF>
   <hasErrorRate>
      <ErrorRate rdf:ID="CD_QoSP_ErrorRate">
         <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">10exp(-5)</Amigo:QC_Value>
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0</Amigo:QC_Metric>
      </ErrorRate>
   </hasErrorRate>
   <hasDataEncryption>
      <DataEncryption rdf:ID="CD_QoSP_DataEncryption">
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0</Amigo:QC_Metric>
         <Amigo:QC_Value
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">AES-128</Amigo:QC_Value>
      </DataEncryption>
   </hasDataEncryption>
   <hasThroughput>
      <Throughput rdf:ID="CD_QoSP_Throughput">
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Kbps</Amigo:QC_Metric>
         <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">512</Amigo:QC_Value>
      </Throughput>
   </hasThroughput>
   <hasSupportedStandard>
      <SupportedStandard rdf:ID="CD_QoSP_SupportedStandard">
         <Amigo:QC_Value
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">WSDL 1.1</Amigo:QC_Value>
         <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0</Amigo:QC_Metric>
      </SupportedStandard>
```

```
    </hasSupportedStandard>
    <hasAvailability>
      <Availability rdf:ID="CD_QoSP_Availability">
        <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0.99995</Amigo:QC_Value>
        <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">0</Amigo:QC_Metric>
      </Availability>
    </hasAvailability>
    <hasMTTR>
      <MTTR rdf:ID="CD_QoSP_MTTR">
        <Amigo:QC_Value rdf:datatype="http://www.w3.org/2001/XMLSchema#string">1800</Amigo:QC_Value>
        <Amigo:QC_Metric rdf:datatype="http://www.w3.org/2001/XMLSchema#string">sec</Amigo:QC_Metric>
      </MTTR>
    </hasMTTR>
</Amigo:Service>
```

### 4.2.4 Event-based service description

The goal of event-based service description is to define semantic services with event capabilities. This kind of service is needed because entities are heterogeneous in terms of their functionalities and mechanisms to produce and consume data.

Event capabilities can be defined in terms of basic operations: subscription, publication, and notification. These operations correspond to standard remote calls. From a higher viewpoint, programmers only express the communication mechanism (i.e., event or command), the direction (i.e., input or output) and the type of the event (i.e., the type of the value which is embedded in the event).

Our system will manage the event throughout its lifecycle and use the command operation as is. While commands are used for synchronous communication between services, events can be used for asynchronous communication. It is important to provide both because, in a ubiquitous environment, deployed services use heterogeneous communication modes (*i.e.*, synchronous and asynchronous). Event design patterns for asynchronous communication rely on several low-level synchronous operations, i.e. subscription and notification. Providing a high-level viewpoint prevents programmers from dealing with these low-level operations when it is unnecessary. Semantic services and their instances can then be separately developed. Indeed, developers only express the communication means, command or event, of semantic services in a typed manner. Typed events enable safer service composition. Valid compositions can then be ensured, i.e. event consumers handle properly received events from event producers.

In our approach, each event is uniquely associated with a type (and vice versa). Consider the following types: *Luminosity*, *Temperature* or *Availability*. One can define an event for each of these types; the defined event then inherits from the semantics of the corresponding type.

An event-based service is described like other regular services in OWL and is associated with a special profile. The profile process indicates if the service consumes or generates events. To do so, it uses one of the following composite processes described in the *events.owl* ontology: *EventOutput* or *EventInput* (see Figure 4-1). This ontology must be imported when a new event-based service description is defined.

In addition, the profile must declare a unique *hasParameter* property. This property defines the type of values associated to an event. The URI of a previously defined datatype is used to define the *hasParameter* property.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:p1="http://www.owl-ontologies.com/Events.owl#"
   xmlns:j.1="http://www.daml.org/services/owl-s/1.1/Process.owl#"
   xmlns:j.0="http://amigo.gforge.inria.fr/owl/Amigo-S.owl#"
   xmlns:j.2="http://www.daml.org/services/owl-s/1.1/Service.owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
```

```
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:j.3="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://www.owl-ontologies.com/Events.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://amigo.gforge.inria.fr/owl/Amigo-S.owl"/>
  </owl:Ontology>
  <j.1:CompositeProcess rdf:ID="EventInput"/>
  <j.1:CompositeProcess rdf:ID="EventOutput"/>
</rdf:RDF>
```

*Figure 4-1: The events.owl ontology*

Let us consider as an example a light manager to illustrate how to describe an event-based service (see Figure 4-2). This light manager subscribes to events provided by a light sensor and turns lights on and off depending on the ambient luminosity. Note that light sensors and lights are also considered as services. These three services are described using the Amigo-S vocabulary and the *events.owl* ontology vocabulary.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:j.0="http://www.daml.org/services/owl-s/1.1/Process.owl#"
    xmlns:j.1="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
    xmlns:j.2="http://www.daml.org/services/owl-s/1.1/Service.owl#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    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/LightManagerExample.owl#"
    xmlns:j.3="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xml:base="http://www.owl-ontologies.com/LightManagerExample.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.owl-ontologies.com/Events.owl"/>
  </owl:Ontology>
  <j.0:ResultVar rdf:ID="Luminosity">
    <j.0:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    ></j.0:parameterType>
  </j.0:ResultVar>
  <j.2:Service rdf:ID="Light">
    <j.2:presents>
      <j.3:Profile rdf:ID="OnOffProfile">
        <j.3:has_process>
          <j.0:CompositeProcess rdf:ID="OnOffProcess">
            <j.0:composedOf>
              <j.0:Any-Order rdf:ID="OnOff">
                <j.0:components>
                  <j.0:ControlConstructBag rdf:ID="OnOffComponent">
                    <j.1:first>
                      <j.0:Perform rdf:ID="Off">
                        <j.0:process>
                          <j.0:AtomicProcess rdf:ID="OffProcess"/>
                        </j.0:process>
                      </j.0:Perform>
                    </j.1:first>
                    <j.1:first>
                      <j.0:Perform rdf:ID="On">
                        <j.0:process>
                          <j.0:AtomicProcess rdf:ID="OnProcess"/>
                        </j.0:process>
                      </j.0:Perform>
                    </j.1:first>
                  </j.0:ControlConstructBag>
                </j.0:components>
              </j.0:Any-Order>
            </j.0:composedOf>
          </j.0:CompositeProcess>
        </j.3:has_process>
      </j.3:Profile>
    </j.2:presents>
  </j.2:Service>
```

```
<j.2:Service rdf:ID="LightManager">
  <j.2:presents>
    <j.3:Profile rdf:ID="LuminosityEventInputProfile">
      <j.3:has_process rdf:resource=
        "http://www.owl-ontologies.com/Events.owl#EventInput"/>
      <j.3:hasParameter rdf:resource="#Luminosity"/>
    </j.3:Profile>
  </j.2:presents>
</j.2:Service>
<j.2:Service rdf:ID="LightSensor">
  <j.2:presents>
    <j.3:Profile rdf:ID="LuminosityEventOuputProfile">
      <j.3:hasParameter rdf:resource="#Luminosity"/>
      <j.3:has_process rdf:resource=
        "http://www.owl-ontologies.com/Events.owl#EventOutput"/>
    </j.3:Profile>
  </j.2:presents>
</j.2:Service>
</rdf:RDF>
```

*Figure 4-2: LightSensor, Light and LightManager service descriptions*

To summarize, our approach enables high-level development, thanks to event abstraction and separation of concerns, and to the separation between semantic service description and service instance. A semantic discovery process enables finding event-based service instances by searching with event-based semantic descriptions.

## 4.3 How to register a semantic service with the service repository

In order to make services available to the Amigo home environment, we must register the services with the service repository. The service repository will typically be hosted on a server machine within the networked home environment, though the services themselves may be hosted on any device within the environment.

The repository can be discovered via the dynamic discovery mechanisms provided by the OSGi programming framework, please refer Chapter 2 for examples of using dynamic discovery.

To register an Amigo-S service with the service repository, we can call the ‘addService’ method:

```
addService(String serviceDescriptionURI, String wsdlGroundingURI);
```

The addService method takes two parameters – the first is the URI of the Amigo-S description of the service, the second is the URI of the service’s grounding, which in this example points to a WSDL file. A URI representing the service instance is returned by the method, and can be used to identify the service instance within the service repository.

A service can be removed from the repository by calling the ‘removeService’ method:

```
removeService(String serviceURI);
```

A special case for service registration is legacy services such as UPnP-based services. Here, no explicit service registration is required. When requests are made for such services, the service repository exploits the Amigo interoperability framework to discover legacy services at runtime.

## 4.4 Resources

[D2.1] Amigo D2.1 Specification of the abstract system architecture. Available on-line at http://www.hitech-projects.com/euprojects/amigo/deliverables/Amigo_WP2_D2.1_v10%20final.pdf.

[D3.1b] Amigo D3.1b Detailed design of the Amigo middleware core: Service specification, Interoperable middleware core. Available on-line at: http://www.hitech-projects.com/euprojects/amigo/deliverables/Amigo_WP3_D31b_v1.0.pdf

[D3.2] Amigo D3.2 Amigo middleware core: Prototype implementation & documentation. Available on-line at: http://www.hitech-projects.com/euprojects/amigo/deliverables/amigo-d3.2-final.pdf.

[Eyeball] Eyeball: a tool for checking RDF/OWL for common problems. See: http://jena.sourceforge.net/Eyeball/.

[OWL-S] OWL-S: Semantic Markup for Web Services. See: http://www.w3.org/Submission/OWL-S/.

[Protégé] The Protégé Ontology Editor and Knowledge Acquisition System. See: http://protege.stanford.edu/.

# 5 How to develop a semantic service-based application

## 5.1 Overview

This chapter presents a set of 'how-to's' for the Amigo application developer, describing how to create applications that incorporate semantic services. We support a comprehensive approach to service description (which was presented in Chapter 4), discovery, composition, adaptation and execution. Our solution includes a number of features an application developer can use when discovering and composing the services to be used by an application, including: for the former, context-aware and quality-of-service based service discovery, for the latter, workflow-based service composition, strategy-based service composition and event-based service composition. Currently, these features are provided as distinct alternatives; however, they will be integrated in later releases.

### 5.1.1 Objectives and principles

#### 5.1.1.1 Workflow-based service composition

We have developed an approach to Service Description – Service Discovery, workflow-based Composition, Adaptation and Execution, which we call SD-SDCAE. The SD-SDCAE solution aims to enable user applications to exploit services deployed in the Amigo home. In the static case, we know in advance which single or multiple services we need to invoke or to compose. We also know the interfaces and the behaviour (i.e., workflow) of these services. These services may be looked up by name and invoked employing the basic service discovery and service interaction (see Chapter 2). However, in the dynamic case, we do not know in advance which services to employ nor their exact interfaces and behaviours. We thus rely on discovery of services based on the semantics of required functionalities.

For this, both our "abstract" request (since we do not know in advance the services that we will finally employ) and the available provided services are semantically described. Our request is described in the form of a task, which is an abstract workflow. Then, we carry out: semantic service discovery, filtering on QoS and context properties as required; service workflow composition if no single service satisfies our request but the composite usage of several services does; and adaptation of our "abstract expectation" to the available service(s). Finally, we execute the "adapted expectation" invoking the single or multiple composed services.

#### 5.1.1.2 Strategy-based service composition

The strategy-based service composition framework is an example of applying design patterns to an adaptive service composition problem. The composition logic can be tested interactively with the VantagePoint tool [D3.3] which is based on semantic modelling of intelligent homes.

#### 5.1.1.3 Context-aware service discovery

A client interested in accessing a service obtains information about the existence of a service, its applicable parameters, and terms, through service discovery. The existing service discovery protocols, such as WS-Discovery, match services considering only the keywords from the user's query and the terms in the service descriptions. These protocols do not consider the context information of the services and clients.

Context-aware computing is a paradigm closely related to mobile computing. Mobile clients usually prefer using services based on several context parameters such as location, time, user identity and profile, device capabilities, etc. This indicates that the client and services context

information influences the quality of the service matching and therefore should be taken into account in the service matching process itself, which is the main purpose of the Context Aware Service Discovery service:

*To take into account different types of context from the client (looking for services) as well as from the services (that are potential matches to the clients' service discovery request) in order to improve the matching results.*

The specific type of context information to take into account depends on the service discovery request from the client (e.g. does it want the closest or the cheapest service).

The objective of the Context Aware Service Discovery (CASD) mechanism therefore is to provide application developers with the means to make the composition of the services and programs which they develop context-aware. This in turn means that the mechanisms used to discover services and applications (service discovery) have to be made context-aware.

#### 5.1.1.4 Quality of service-aware service discovery

A plethora of services will eventually be deployed in the Amigo home. Many of these services will be offering similar functionality. For serving a specific service request of a user, the Amigo home middleware should be able to select the most suitable one among services with similar functionality and similar IOPE (Inputs-Outputs-Preconditions-Effects) parameters, all addressing the user requirements. Thus, in the context of QoS-aware service selection, a client submits its service request to the QoS-Aware Service Selection Tool (QASST). The QASST uses the base service discovery mechanism to identify the services matching the functional requirements of the service request received. Then, it retrieves the user's QoS preferences from the User Modelling and Profiling Service (UMPS) of the WP4 Intelligent User Services layer. Subsequently, the matching mechanism of QASST takes control, filters out the services that do not address the user's QoS requirements and finally selects the most appropriate one based on an efficient service selection algorithm. The selected service is then invoked in a completely transparent to the user manner through the middleware mechanisms.

In a nutshell, QASST is provided by the Amigo Middleware and implements QoS-aware service selection mechanisms adequate for services concerning in-home activities, for example content delivery services that reside on Amigo home devices, in order for the users to be provided with services that address their QoS requirements as much as possible.

#### 5.1.1.5 Event-based service composition

Our goal is to provide a Domain-Specific Language (DSL), namely Pantachou, to ease service development. This scripting language will provide high-level abstractions to heterogeneous communication modes, i.e., command and event.

Programmers use a small yet expressive scripting language to develop applications. These applications can communicate via command and event operations. They are compiled to standard Java code and run smoothly in the Amigo framework.

### 5.1.2 Features

#### 5.1.2.1 Workflow-based service composition

The features of developing an application using Amigo-S and SD-SDCAE include:

- A user task provides an abstract description of the required capabilities of an application.
- The required capabilities of a task can be identified by the semantic of the capability and of its inputs and outputs.

- An application may invoke capabilities of varying complexity, from lightweight atomic calls to complex, interleaved conversations (i.e., service workflows), for one or many tasks from within the same application.
- The conversation (workflow) of a required capability of a task is automatically reconstituted by weaving together the conversations (workflows) of the provided capabilities of the available services.
- The resulting (composed) service is generated as an executable ActiveBPEL bundle and automatically deployed.

#### 5.1.2.2 Strategy-based service composition

The strategy-based service composition framework is developed as platform independent UML model. An implementation of the framework that is integrated with the VantagePoint tool is provided for testing the compositions at development time.

#### 5.1.2.3 Context-aware service discovery

Based on the motivation in the previous section, the design of a Context-Aware Service Discovery (CASD) mechanism should be able to:

A. Determine the most suitable service by taking into account the context information of both the service and client, and the ranking algorithm as requested by the client (Closest, cheapest, etc).

B. In the event of a context change or the appearance of new services, if a more suitable service is found, to notify the client of this more appropriate service by means of a persistent service discovery mechanism (if the client wishes to be notified of this).

C. Use ontologies for context representation and processing during the service matching process, to leverage results from other parts of the Amigo project, such as the Context Management Service (CMS).

In the CASD model, every service and client may have one (or possibly more) context sources. The context source provides context information about the associated service or client.

The services register with the service directory (or other suitable service discovery mechanisms) so they can be discovered. The assumption is that a service has knowledge about its context sources, so that it can provide references to these context sources when the service registers with the discovery mechanism in a way that is best suited for that particular mechanism.

For basic discovery mechanisms it is usually possible to set general name/value properties, such as with WS-Discovery. In this case a specifically named property of the service will be set, with a value pointing to the context source providing information about the service, for example:

ContextSourceURL=http://my.fqdn.org:8080/context/locationCS.

By providing such a reference to a Context Source, which can provide context for a particular service, the CASD is able to retrieve the context it wants at run-time from that context source and use it in its matching process.

Since the context of the client may also be needed for producing a ranking based on context (e.g. to find the closest service, the location of the client should be known) the client should also specify a reference to a context source in the request it makes to the CASD service.

The ranking that the client requests (closest, cheapest, etc.) puts requirements on the type of context that the different context sources should be able to provide.

If the client makes a request for a suitable service with the CASD service, it includes a service description of the service the client is looking for. This description is in a form appropriate for the service discovery mechanism used (e.g. WS-Discovery). The request also includes the ranking algorithm (closest, cheapest, etc.) that the CASD service should apply to the list of matching services. The CASD service retrieves the services matching the service type specified by the client after querying the basic service discovery mechanism. Such services are referred to as basic matching services. The CASD service then collects the context information of the basic services and of the client. It then further filters or ranks the basic services based on that context information to return the most suitable service(s) to the client.

To provide an overview of what a developer can do with CASD, we explain a (WP5) application referred to as the Crisis Response Application (CRA), which uses CASD. With the help of this example, we will explain the prototype implementation and documentation for the CASD.

The CRA is responsible to act whenever an emergency situation is detected. The CRA will be made aware of any crisis situation in the home by subscribing to the Awareness and Notification Service (ANS) with the specific rules to be aware of certain emergency conditions. The ANS then notifies the CRA whenever the rule condition is satisfied.

For example, CRA specifies a rule in ANS to be notified when an intruder is detected. The ANS will then subscribe to the Anti-Intrusion Service Context Source to provide certain context data to ANS that ANS needs for executing that specific rule.

Once the CRA receives a notification from ANS that some type of emergency has occurred, it will use the CASD service to request the proper type of service to contact for this particular type of emergency, taking into account the context information of the emergency (e.g. time, type of event, location) and of the Emergency Response Service (e.g. location, speed).

The type of emergency will determine the ranking algorithm requested from the CASD. What this means in practice is that the CRA will for example ask CASD to provide the ‘closest’ (medical emergency) service in case of a medical emergency, and the ‘cheapest’ (plumbing) service in case of a minor water leak. So depending on the type (and urgency) of the emergency that has occurred, different types of context of both the service and client (the CRA) will be taken into account.

#### 5.1.2.4 Quality of service-aware service discovery

The Quality of Service Aware Service Selection Tool (QASST) provided by the Amigo middleware supports a range of commonly required quality of service parameters including throughput, cost, latency, response time, error rate, jitter, data encryption, and accessibility. In this respect, it aims to select the services that address the user’s QoS preferences in the most suitable manner, as well as perform a QoS semantic matching to select the most appropriate QoS-aware services to be integrated in a QoS-aware composed application.

#### 5.1.2.5 Event-based service composition

Pantachou provides abstractions to compose services and make them interact with each other. These interactions can be defined in terms of commands or events. For events, Pantachou provides some syntactic notations to receive and send events. They allow programmers to concentrate on the goal of their applications without dealing with implementation details.

A notion of “behaviour” describes the operations that must be performed when an event is received. This behaviour can be dynamically changed to enable application reconfiguration.

#### 5.1.2.6 Future integration

Currently, these different styles of composition and enhanced service discovery are presented as distinct approaches. A unified, integrated interface is currently being developed that will apply globally Amigo-S service descriptions, use the common semantic service repository introduced in Chapter 4, and allow client requests to use a combined, uniform *Dispatcher* API. This Dispatcher will cleverly relay a client request to the appropriate discovery/composition approach.

### 5.1.3 Assessment

#### 5.1.3.1 Workflow-based service composition

The advantages of developing an application using Amigo-S and SD-SDCAE include:

- By describing a task in an abstract way, we are not bound to any particular remote service in terms of the capabilities provided or the specific orchestration of these capabilities, increasing the availability and promoting interoperability of the potentially matching services.
- Identifying a task's required capabilities by their semantic allows concrete details of the services' provided capabilities used in a composition to be reliably and automatically adapted to the needs of the required capabilities in the absence of an exact match.
- By supporting a variety of depths of complexity of composition, the SD-SDCAE mechanism allows the application developer to control the level of runtime overhead consumed by the composition process to match the needs of the Amigo application.
- Complex conversations can be automatically and reliably composed, while offering fine-grained control over the placement of capabilities in the task, and guaranteeing that the data and control dependencies of each of the provided capabilities are preserved.
- Orchestration of the execution of the composed service is handled automatically by the ActiveBPEL execution engine.
- By separating the composition of service from its invocation, once a new, composed service is created, it can be invoked many times without incurring the cost of composition again.

#### 5.1.3.2 Strategy-based service composition

The strategy-based composition framework provides means to describe reasoning related to service composition in a declarative way. The composition logic can be interactively tested against device libraries providing Amigo-S services with simulated scenarios using the VantagePoint tool.

#### 5.1.3.3 Context-aware service discovery

The Context-Aware Service Discovery (CASD) augments traditional service discovery mechanisms with context-awareness. This means that the context information of both the services and the client is taken into account in determining the list of services that are returned to the client.

One of the features of CASD is that the client need not worry about selecting the most suitable service from the large set of services returned by the service discovery mechanism, since the CASD will provide a ranking of the services based on the context of services and client. The specific ranking method used is determined by the client itself in its request to the CASD service; so the returned list is optimised according to the wishes of the client.

The CASD service also introduces the concept of persistent service discovery, where the clients will be notified whenever the resulting (order of the) list changes, e.g. because the context of some of the services changes, or new services become available. The only change

with respect to the standard mechanism is that the client also specifies a call-back reference in its request to the CASD service, which the CASD service will use to notify the client whenever the result of the request changes.

This mechanism promises to simplify the design of clients in pervasive environments as they need not actively search for the best services when their context changes. This added simplicity is due to the fact that they will be dynamically notified of better service matches as they become available, meaning that clients will not have to adopt a 'polling style' service discovery.

#### 5.1.3.4 Quality of service-aware service discovery

QoS-aware Service Selection Tool (QASST) is used in order to select a single service from a set of services, all addressing the functional capabilities of a user request. Semantic matching mechanisms are applied, which filter out the services that do not address user's QoS preferences, and an efficient selection algorithm is used in order to identify the service that best addresses the submitted request. Both the matching mechanisms and the selection algorithm are based on the robust Amigo QoS semantic framework, which enables users to create multiple QoS profiles for different service types (e.g. VideoDelivery, Gaming) and facilitates the exact semantic matching. Furthermore, for each service type, the user is able to specify not only value thresholds for each QoS parameter (i.e. providing upper, lower or both value constraints), but also the level of significance of this particular QoS parameter. The selection process is based on a straightforward and low-complexity service filtering algorithm and a light-weight selection scheme that takes into consideration all the parameters above. The QASST provides user-friendly interfaces and, once the user QoS preferences have been specified, it selects the service that best addresses these preferences in a completely transparent manner. Furthermore, the functionality of the QASST is extended to support the integration of QoS-aware services in a composed application. One of the major innovations in the proposed approach is that it establishes full scale QoS-awareness, an aspect rarely addressed in Ambient Intelligence, while this is accomplished in a lightweight and transparent manner.

#### 5.1.3.5 Event-based service composition

Pantachou is dedicated to event-based service composition. It provides high-level abstractions to express event-based asynchronous communication between services. These high-level abstractions will reduce development time thanks to concise and easy to write applications. Programmers will concentrate on *what* their applications should do instead of *how* they do it. Pantachou reuses semantic service description and provides operations to compose semantic services. Pantachou applications do not rely on specific service instances but on specific service functionalities, which have been expressed in semantic descriptions. This high-level composition enables deployment of Pantachou applications in several ubiquitous environments with different services. The only requirement for such a deployment is that the new targeted environment provides semantically equivalent service instances.

Moreover, compared to general-purpose languages, these DSL programs will be safer as Pantachou enables more code verification. The Pantachou compiler could for instance ensure valid compositions. To do so, it would analyze Pantachou applications and match them against their corresponding semantic service descriptions. Pantachou applications and composed event-based services must be compliant with their corresponding semantic descriptions. Moreover, these semantic descriptions must be compliant with respect to the composition expressed by the Pantachou application logic.

## 5.2 How to develop an application that integrates complex service workflows

In this section, we will walk through creating a task description in Amigo-S, and how the task is realised by composing available Amigo-S services. This will include writing the task description, registering the Amigo-S services developed in Chapter 4, calling the SD-SDCAE mechanism to realise the task, and demonstrating how the resulting service can be executed from Java code.

Before we begin creating our task description, it may be useful to examine Figure 5-1, which gives an impression of the overall SD-SDCAE service composition process.

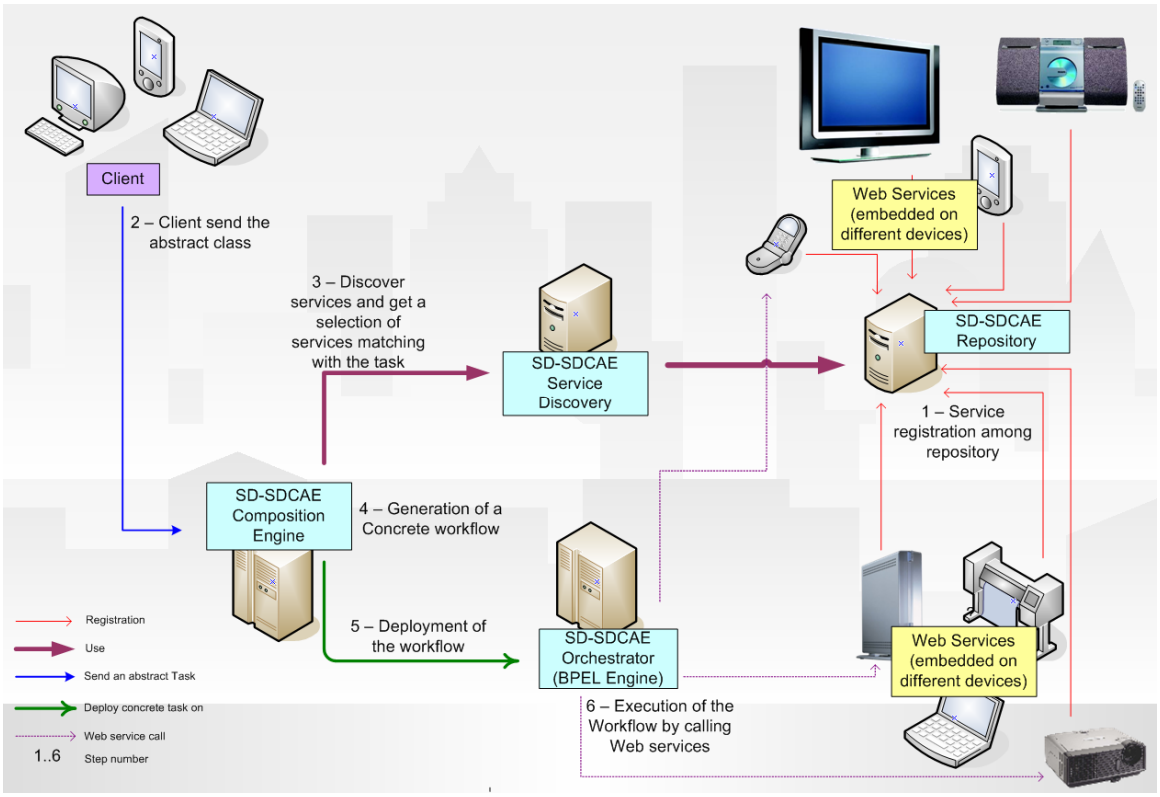


*Figure 5-1: An overview of the steps involved in the SD-SDCAE composition process*

The first step of developing an SD-SDCAE application is to create a task description. The task specifies the set of required capabilities the application needs to run. The task is described abstractly, it is not yet bound to any remote service either in terms of the set of required capabilities to be provided by remote services or in the orchestration of these capabilities to be matched with the conversations (workflows of capabilities) of the remote services.

Continuing the example from Chapter 4, we will create a task description for a multimedia player application. The task description can be created using the application developer's tools of choice. As a guide, the source OWL XML of the multimedia player is provided below:

```
<?xml version="1.0"?>
<!DOCTYPE uridef[
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY process "http://www.daml.org/services/owl-s/1.1/Process.owl">
  <!ENTITY objList "http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl">
  <!ENTITY wsdl
    "http://www.hitech-projects.com/euprojects/amigo/MultimediaRenderer.wsdl">
```

```
]>
<rdf:RDF
  xmlns:lang="http://www.hitech-projects.com/euprojects/amigo/Amigo-S.owl#"
  xmlns:capabilities=
    "http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#"
  xmlns:process="http://www.daml.org/services/owl-s/1.1/Process.owl#"
  xmlns:service="http://www.daml.org/services/owl-s/1.1/Service.owl#"
  xmlns:profile="http://www.daml.org/services/owl-s/1.1/Profile.owl#"
  xmlns:grounding="http://www.daml.org/services/owl-s/1.1/Grounding.owl#"
  xmlns:objList="http://www.daml.org/services/owl-s/1.1/generic/ObjectList.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  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:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns="http://www.hitech-projects.com/euprojects/amigo/MultimediaPlayer.owl#"
  xml:base="http://www.hitech-projects.com/euprojects/amigo/MultimediaPlayer.owl"
  xmlns:wsdl="http://www.hitech-projects.com/euprojects/amigo/MultimediaPlayer.wsdl">

  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource=
      "http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl"/>
    <owl:imports rdf:resource=
      "http://www.hitech-projects.com/euprojects/amigo/Amigo-S.owl"/>
  </owl:Ontology>
```

These set up the required namespaces and imports for a task. Next, we describe the required capabilities needed by the application in the task's service profile:

```
<service:Service rdf:ID="MultimediaPlayerService">
  <service:presents>
    <capabilities:ServiceProfile rdf:ID="MultimediaPlayerProfile">

      <!-- Required capabilities -->
      <lang:hasRequiredCapability>
        <capabilities:ListMedia rdf:ID="ListMediaCapability">
          <lang:hasConversation rdf:resource="#ListMediaConversation"/>
          <lang:hasOutput rdf:resource="#ListMediaOutput"/>
        </capabilities:ListMedia>
      </lang:hasRequiredCapability>

      <lang:hasRequiredCapability>
        <capabilities:ListMostPlayedMedia rdf:ID="ListMostPlayedMediaCapability">
          <lang:hasConversation rdf:resource="#ListMostPlayedMediaConversation"/>
          <lang:hasInput rdf:resource="#ListMostPlayedMediaInput"/>
          <lang:hasOutput rdf:resource="#ListMostPlayedMediaOutput"/>
        </capabilities:ListMostPlayedMedia>
      </lang:hasRequiredCapability>

      <lang:hasRequiredCapability>
        <capabilities:PlayMedia rdf:ID="PlayMediaCapability">
          <lang:hasConversation rdf:resource="#PlayMediaConversation"/>
          <lang:hasInput rdf:resource="#PlayMediaInput"/>
          <lang:hasOutput rdf:resource="#PlayMediaOutput"/>
        </capabilities:PlayMedia>
      </lang:hasRequiredCapability>

      <!-- Sub-capabilities -->
      <lang:hasCapability>
        <capabilities:Login rdf:ID="LoginCapability">
          <lang:hasConversation rdf:resource="#LoginConversation"/>
          <lang:hasInput rdf:resource="#LoginInput"/>
        </capabilities:Login>
      </lang:hasCapability>

      <lang:hasCapability>
        <capabilities:CreateMostPlayedMediaList rdf:ID=
          "CreateMostPlayedMediaListCapability">
          <lang:hasConversation rdf:resource=
            "#CreateMostPlayedMediaListConversation"/>
          <lang:hasOutput rdf:resource="#CreateMostPlayedMediaListOutput"/>
        </capabilities:CreateMostPlayedMediaList>
      </lang:hasCapability>
```

```
    <lang:hasCapability>
      <capabilities:SetMedia rdf:ID="SetMediaCapability">
        <lang:hasConversation rdf:resource="#SetMediaConversation"/>
        <lang:hasInput rdf:resource="#SetMediaInput"/>
      </capabilities:SetMedia>
    </lang:hasCapability>

    <lang:hasCapability>
      <capabilities:Play rdf:ID="PlayCapability">
        <lang:hasConversation rdf:resource="#PlayConversation"/>
        <lang:hasOutput rdf:resource="#PlayOutput"/>
      </capabilities:Play>
    </lang:hasCapability>

    <service:presentedBy rdf:resource="#MultimediaPlayerService"/>

    <profile:textDescription rdf:datatype=
      "http://www.w3.org/2001/XMLSchema#string">
      Example multimedia player service.
    </profile:textDescription>

    <profile:serviceName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      MultimediaPlayer Service
    </profile:serviceName>
  </capabilities:ServiceProfile>
 </service:presents>
</service:Service>
```

Following this, we provide the structure of the conversations of the task's capabilities:

```
<!-- Conversations -->
<process:AtomicProcess rdf:ID="ListMediaConversation">
 <process:hasOutput>
  <process:Output rdf:ID="ListMediaOutput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#MediaList
   </process:parameterType>
  </process:Output>
 </process:hasOutput>
</process:AtomicProcess>

<process:CompositeProcess rdf:ID="ListMostPlayedMediaConversation">
 <process:hasInput>
  <process:Input rdf:ID="ListMostPlayedMediaInput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#User
   </process:parameterType>
  </process:Input>
 </process:hasInput>

 <process:hasOutput>
  <process:Output rdf:ID="ListMostPlayedMediaOutput">
   <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
    http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#MediaList
   </process:parameterType>
  </process:Output>
 </process:hasOutput>

 <process:composedOf>
  <process:Sequence>
   <process:components>
    <process:ControlConstructList>
     <objList:first>
      <process:Perform rdf:ID="LoginPerform">
       <process:process rdf:resource="#LoginCapability"/>
       <!-- Parameter Bindings -->
      </process:Perform>
     </objList:first>
     <objList:rest>
      <process:ControlConstructList>
       <objList:first>
        <process:Perform rdf:ID="CreateMostPlayedMediaListPerform">
         <process:process rdf:resource="#CreateMostPlayedMediaListCapability"/>
```

```
            <!-- Parameter Bindings -->
           </process:Perform>
          </objList:first>
          <objList:rest rdf:resource="&objList;#nil"/>
         </process:ControlConstructList>
        </objList:rest>

      </process:ControlConstructList>
    </process:components>
   </process:Sequence>
  </process:composedOf>
</process:CompositeProcess>

<process:AtomicProcess rdf:ID="LoginConversation">
  <process:hasInput>
    <process:Input rdf:ID="LoginInput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#User
      </process:parameterType>
    </process:Input>
  </process:hasInput>
</process:AtomicProcess>

<process:AtomicProcess rdf:ID="CreateMostPlayedMediaListConversation">
  <process:hasOutput>
    <process:Output rdf:ID="CreateMostPlayedMediaListOutput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#MediaList
      </process:parameterType>
    </process:Output>
  </process:hasOutput>
</process:AtomicProcess>

<process:CompositeProcess rdf:ID="PlayMediaConversation">
  <process:hasInput>
    <process:Input rdf:ID="PlayMediaInput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#Media
      </process:parameterType>
    </process:Input>
  </process:hasInput>

  <process:hasOutput>
    <process:Output rdf:ID="PlayMediaOutput">
      <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
        http://www.w3.org/2001/XMLSchema#boolean
      </process:parameterType>
    </process:Output>
  </process:hasOutput>

  <process:composedOf>
    <process:Sequence>
      <process:components>
        <process:ControlConstructList>
          <objList:first>
            <process:Perform rdf:ID="SetMediaPerform">
              <process:process rdf:resource="#SetMediaCapability"/>
              <!-- Parameter Bindings -->
            </process:Perform>
          </objList:first>
          <objList:rest>
            <process:ControlConstructList>
              <objList:first>
                <process:Perform rdf:ID="PlayPerform">
                  <process:process rdf:resource="#PlayCapability"/>
                  <!-- Parameter Bindings -->
                </process:Perform>
              </objList:first>
              <objList:rest rdf:resource="&objList;#nil"/>
            </process:ControlConstructList>
          </objList:rest>

        </process:ControlConstructList>
      </process:components>
    </process:Sequence>
```

```
    </process:composedOf>
  </process:CompositeProcess>

  <process:AtomicProcess rdf:ID="SetMediaConversation">
    <process:hasInput>
      <process:Input rdf:ID="SetMediaInput">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
          http://www.hitech-projects.com/euprojects/amigo/Capabilities.owl#Media
        </process:parameterType>
      </process:Input>
    </process:hasInput>
  </process:AtomicProcess>

  <process:AtomicProcess rdf:ID="PlayConversation">
    <process:hasOutput>
      <process:Output rdf:ID="PlayOutput">
        <process:parameterType rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
          http://www.w3.org/2001/XMLSchema#boolean
        </process:parameterType>
      </process:Output>
    </process:hasOutput>
  </process:AtomicProcess>
</rdf:RDF>
```

Note that no grounding is defined, as the task is an abstract description.

Now that we have a task description for the multimedia player application, and service descriptions for matching services, we are ready to deploy the application.

First, in order to make the services available to the SD-SDCAE mechanism, we must register the services with the service repository. The service repository will typically be hosted on a server machine within the networked home environment, though the services themselves may be hosted on any device within the environment.

The repository can be discovered via the dynamic discovery mechanisms provided by the OSGi programming framework, please refer to Chapter 2 for examples of using dynamic discovery.

For example, to register the MultimediaBrowser service (detailed in Chapter 4), we would call the ‘addService’ method like this:

```
String multimediaBrowserURI = repository.addService(multimediaBrowserServiceDescriptionURI,
                                    multimediaBrowserWsdlGroundingURI);
```

Now that the services are registered, we can call SD-SDCAE to realize our task. To do this, we first discover the SD-SDCAE composition engine, again using the discovery mechanisms of the OSGi programming framework. Then, we must execute the following code:

```
 Boolean success = compositionEngine.deployAsService(taskURI);
CapabilityInvoker invoker = null;
if (success.equals(Boolean.TRUE)) {
  invoker = compositionEngine.getCapabilityInvoker(taskURI);
}
```

The deployAsService method takes the URI of the task description as a parameter. When this method is called, the SD-SDCAE attempts to realise the required capabilities of the task by composing the provided capabilities of the available services. If a composition is found that satisfies the requirements of the task, SD-SDCAE generates an executable BPEL bundle for each required capability of the task. These bundles are then deployed on a suitable host machine. Typically, this host will be a server machine within the home environment, though the application which invokes these bundles may be hosted on any device within the environment. Note that the application is shielded from these implementation details – in order to invoke the required capabilities of the task, we only need check that the composition succeeded, and obtain the capability invoker for the task, as in the code snippet above.

Once we have obtained the capability invoker for the task, we can invoke any of the task capabilities as desired. For example, the code for the multimedia player application makes the following invocations:

```
String media = null;
if (mode.equals(Mode.SHUFFLE)) {
  List<String> allMedia = invoker.invoke("ListMediaCapability");
  media = allMedia.get(random.nextInt(allMedia.size()));
}
else if (mode.equals(Mode.SHUFFLE_MOST_PLAYED)) {
  List<String> mostPlayedMedia =
    invoker.invoke("ListMostPlayedMediaCapability", "ListMostPlayedMediaInput=" + userID);
  media = mostPlayedMedia.get(random.nextInt(mostPlayedMedia.size()));
}

if (media != null) {
  invoker.invoke("PlayMediaCapability", "PlayMediaInput=" + media);
}
```

This completes our walkthrough of creating and deploying a task using the core language of Amigo-S.

## 5.3 How to develop an application using strategy-based composition and a composition visualisation tool

### 5.3.1 Overview of the composition framework

The composition framework is a set of design patterns and knowledge-based approaches. The goal is to provide a conceptual model described in UML that can be implemented with various programming languages. The prototype is implemented with Java using MDA-based approaches.

The resulting Java framework suits best for checking the context conditions of system before selecting required QoS parameters for the services to be discovered. The use of semantic service discovery and Context Management System is integrated with the framework (for the latter, see the [D4.X] deliverables).

A designer can describe an adaptive composition as a set of hierarchical and prioritized composition rules. The leaf level rules define primitive compositions and support a specific composition strategy. The framework is integrated with VantagePoint [D3.3] so that the composition can be simulated against different application scenarios.

The composition framework can use one or more composition strategies. One of the strategies can be workflow-based, another a user-defined agent-based Java architecture. A composition strategy must support sequential composition of primitive composition fragments into a larger composition.

The composition framework restricts the application to one type of composition strategy at a time, i.e. workflow-based composition cannot be freely combined with Java code.

### 5.3.2 An example scenario

Ambience sharing application may be considered as a context-adaptive extension of traditional person to person visual communication services such as videoconference.

It is composed of several A/V capture and A/V rendering services. Depending on the situation, the service composition system dynamically selects relevant capture and rendering services and establishes a connection (i.e. stream redirection) between them through the A/V relay service.

In the example we consider a static situation where a User activates the Ambience sharing application in a room and the composition is done using the services available in the room.

Later we also discuss the dynamic situation where the user can move from one room to another and the Ambience sharing application uses the services provided by the devices that are available in the room every time when the user moves from one room to another.

### 5.3.3 Simulating service composition logic using the VantagePoint tool

The VantagePoint tool helps an application developer to develop an application composing the available services in different Amigo homes. With VantagePoint the developer can create visualisations of application related scenarios and by editing the model simulate the contextual changes associated with the scenario. A library of devices providing Amigo-S services supports the developer. With these the developer can design one or more intelligent home scenarios and test and verify the resulting composition.

Figure 5-2 shows a visualisation of scenario discussed earlier. There are two houses defined for the scenario with a person each (Roberto and Maria). The house of Roberto is modelled in more detail so that we can test the composition logic with different set of devices providing the required A/V services for the application. The rooms in Maria's house have not been defined in more detail.

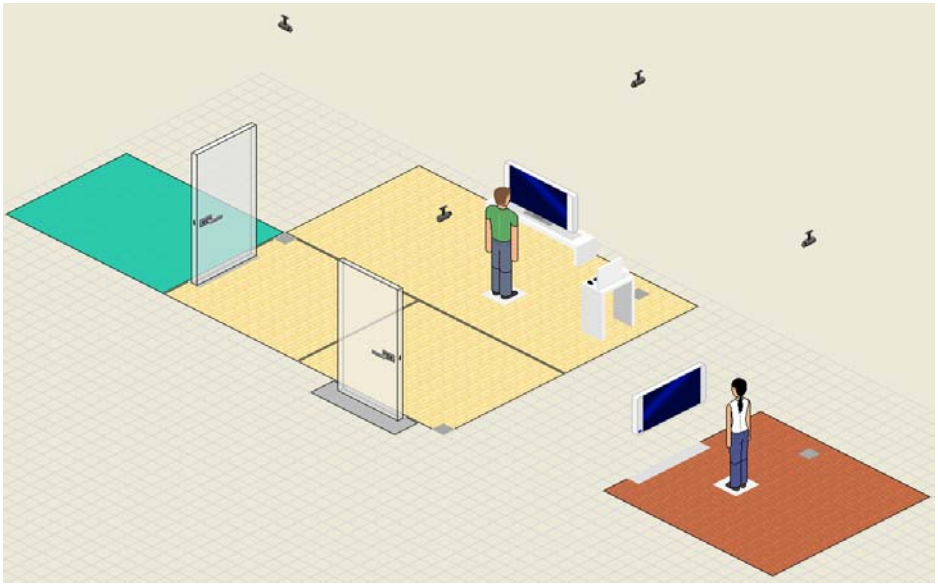

*Figure 5-2: Example scenario visualization in VantagePoint*

The service composition problem for the example scenario is described in Figure 5-3.

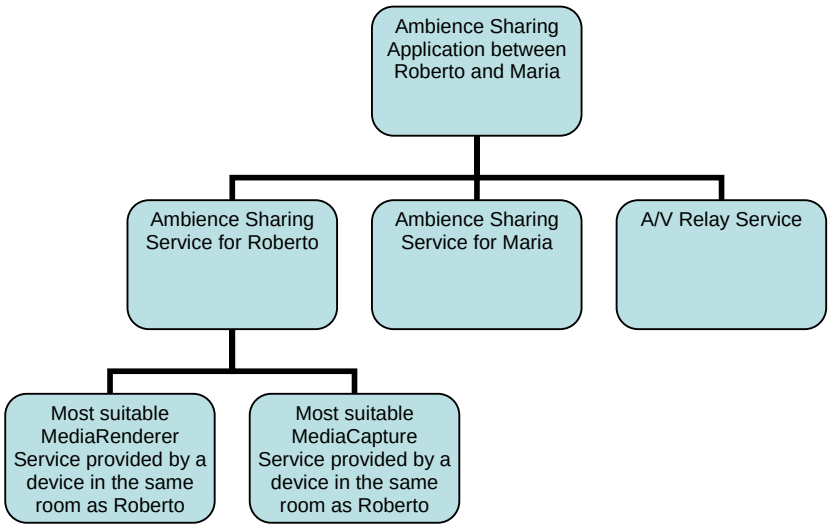


*Figure 5-3: Services to be composed*

The simulation of service composition in VantagePoint is done by using SPARQL queries and listening for VantagePoint events triggered by a user. By implementing the VPEventListener interface and registering as a listener, a Java class can be aware of what happens in the virtual environment. Then queries can be executed according to events such as “item added” or “item removed” to gather relevant data from the VantagePoint model. This data is then used to make the service related computations.

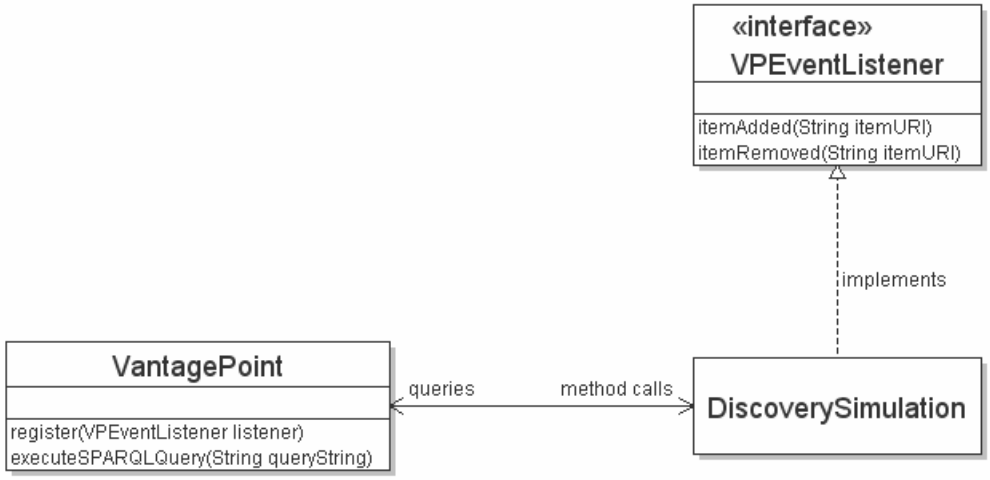


*Figure 5-4: Participating modules in simulation*

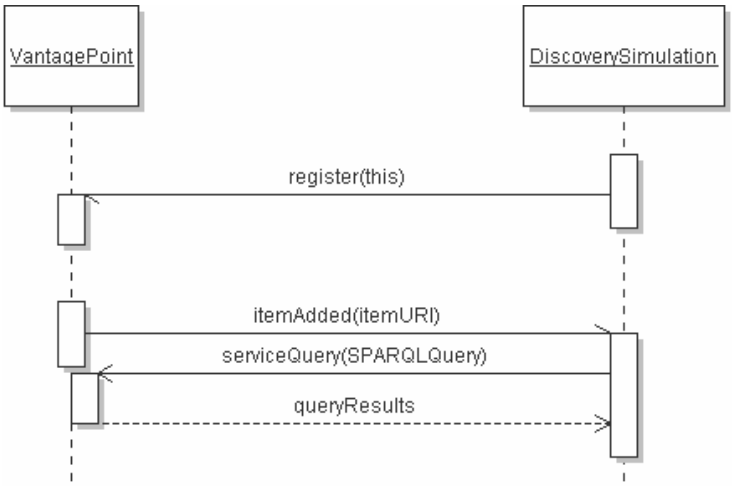


*Figure 5-5: Query sequence*

**Adding new service descriptions to VantagePoint item library**

VantagePoint has an item library which contains predefined items that the user can drag and drop into the virtual environment. Items are basically instances of user specified OWL classes that have icon information attached to them and possibly some extra data, for example service description. This extra data is referenced in the item description. In the example below there is a single item from the item library. It is an instance of OWL class “Speaker”, it uses VantagePoint’s icon set “speakerIconSet1” and references a service instance.

```
<rdf:Description rdf:about="&ItemLibrary;MySpeakerItem">
  <iconSet>speakerIconSet1</iconSet>
  <rdf:type rdf:resource="&Amigo;Speaker"/>
  <Amigo:deploysService rdf:resource="&Amigo;SpeakerServiceInstance"/>
</rdf:Description>
```

The class definitions and the actual service instance have to be imported to the VantagePoint model in order to make the querying work correctly and the service description available. In the case of the example above, the OWL file that specifies the class “Speaker” and instance “SpeakerServiceInstance” should be imported to the VantagePoint model. Importing has to be done manually, at the moment, by adding the desired OWL file references in the WorldModel.owl import list.

After the item library has items that provide services, the service composition simulation can begin.

**Static service discovery**

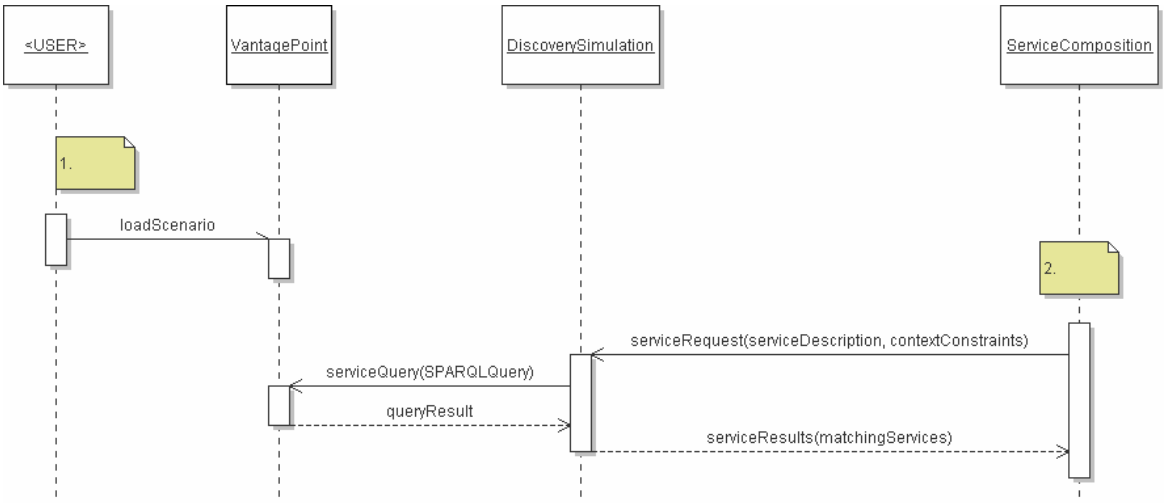


*Figure 5-6: Service discovery simulation*

The Figure 5-6 goes through the sequence of messages and commands that take place in the service discovery simulation. In addition to VantagePoint and the service composition application there are separate classes to simulate the service discovery and context management system, which will be demonstrated later.

1. User loads a model of a house which is the scenario for this simulation. The house model contains the items that provide services for this case.
2. Service composition application makes a service request. Context constraints here mean for example the area from which to search for matching services. The discovery simulation queries VantagePoint for services in an area and gets the results and parses them in to a service description. Then the list of matching services is returned back to the requesting application. This step is the basic example of service discovery.

The place for service registry implementation is between VantagePoint and discovery simulation. Discovery simulation would communicate only with service registry and VantagePoint would be the responsible for updating the registry.

**Dynamic service discovery**

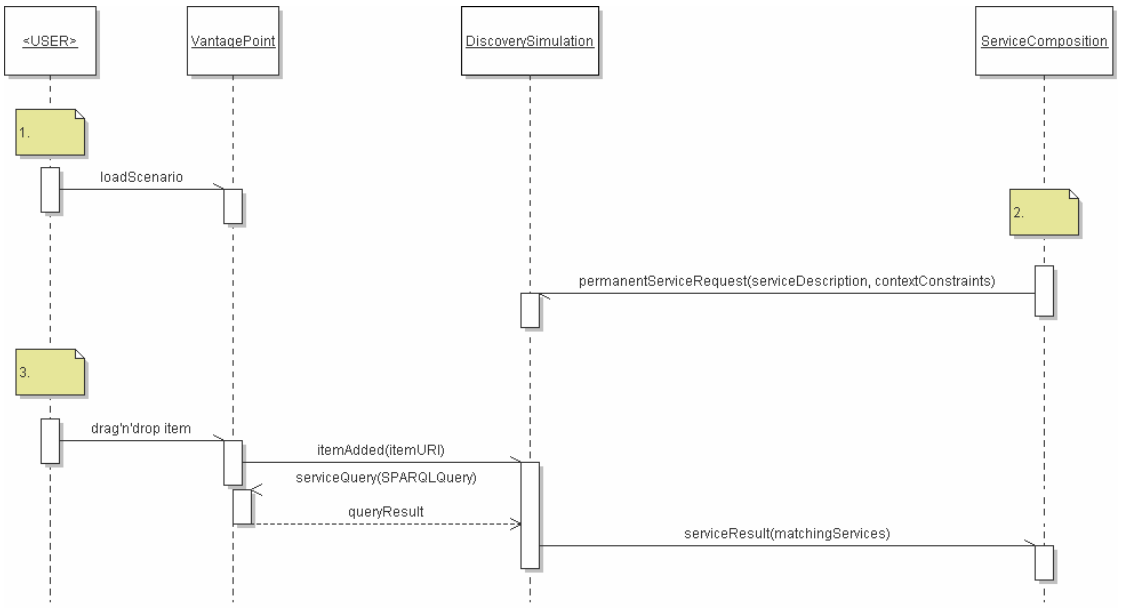


*Figure 5-7: Dynamic discovery of services*

User can add items to the VantagePoint model by dragging and dropping. This user gesture starts the dynamic service discovery sequence. VantagePoint sends the recently added item URI to the applications that are registered as its listeners. Then applications can react to this event by querying the new item about its services.

1. User loads a model of a house.
2. Service composition application makes a permanent service request. This means that it wants to be notified whenever a matching service is discovered. Context constraints could define the area from where to search for matching services.
3. User drops an item to the virtual environment. An “item added” event is generated. Now the discovery simulation queries VantagePoint if the new item should have any services to provide and whether the services match with the permanent service request. Matching service descriptions are then sent to service composition application.

**Simulating context events**

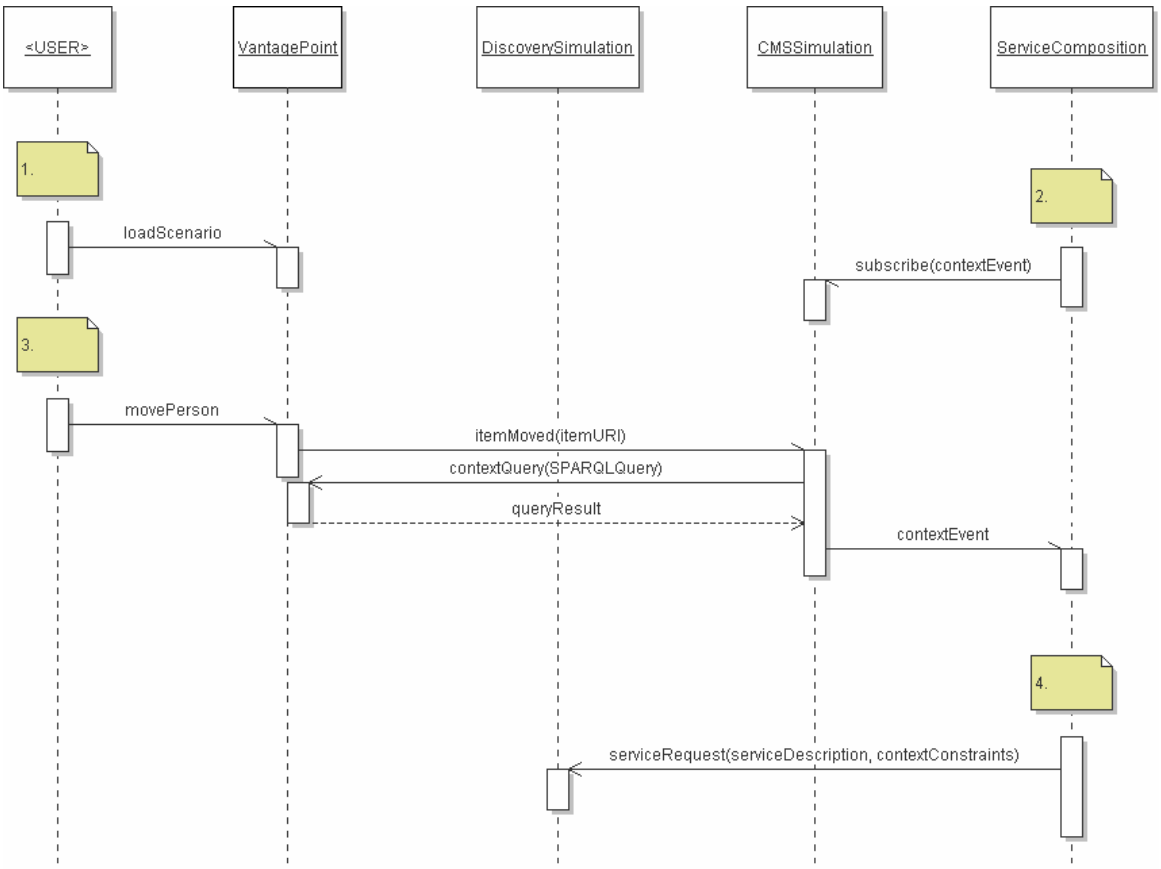


*Figure 5-8: Simulating context events*

By moving items around in the VantagePoint model user creates context events. The ServiceSimulation can be notified about these events so that it can react in appropriate way. The sequence of context events simulation goes as follows.

1. User loads a model of a house which is the scenario for this simulation. The house model contains the items that provide services for this case.
2. Service composition application subscribes a context event. It wants to be notified if this particular context event occurs. In this case the interesting event is person moving from area to another area and the goal is to perform “follow me” –type of tasks.
3. User moves an item representing a person from area to another area. An “item moved” event is generated. Then the CMSSimulation registered to listen to VantagePoint events queries if the item is a person and moved from area to another. If so, it notifies the service composition application for this event.
4. Service composition application reacts to context changes and performs a new service request for the area where the person has moved into.

### 5.3.4 Describing a composition using the ESRR framework

The service composition is described by defining one or more *composition rule* classes that describe the composition logic and by selecting a suitable *composition strategy* to describe how the selected services are composed into task that can be executed. The composition

strategy can be a simple Java code fragment that calls the selected web services, or it can be a new Amigo-S abstract composite service.

The services used by ambience sharing composition logic are:

**MediaRendering**: This service renders A/V streams and can be deployed on every screen in the house.

**MediaCapture**: This is a MultimediaMediaServer service that provides captured A/V streams and can be deployed on every camera and microphone in the house.

**MediaControlPoint:** the role of this service is to obtain streams from A/V capture services and to direct them to A/V rendering services. The service provides an interface for configuring the streams directions (i.e. which capture services to receive from and which rendering services to render to).

For the example scenario we show how to compose PersonAmbience service that can be used by an ambience sharing application to capture the person related A/V stream and render ambience A/V streams provided for another person in other house.

The composition logic is described using hierarchical reasoning. A composition has to first know where the user is in order to be able to discover the suitable MediaRendering and MediaCapture services. We first create a VantagePoint query template to check in what room the person is.

```
"SELECT ?room  WHERE  #INSTANCE# <Amigo.owl#IsLocatedIn> room"
```

This query can be tested in the simulated scenario with VantagePoint query support so that it provides the correct results.

The compositions rule is defined by creating a subclass of the ESSR.Composition class. The overall logic of reasoning is given as a comment:

```
/* COMPOSITION PersonAmbianceSharing for User1
R1: CONDITION User1 is located in Room
        ACTION COMPOSE PersonAmbienceSharingInRoom(Person,Room)
        DEFAULT: do nothing*/

public PersonAmbianceSharing extends ESRR.Composition {
        public PersonAmbianceSharing (String User1)
        {
                R1 is new ESRR.Rule;
                R1.Condition.Query =
                        "SELECT ?room  WHERE" + User1 + "<Amigo.owl#IsLocatedIn> room" ;
                R1.Action = new ESRR.CompositeStrategy();
                R1.Action.add(new PersonAmbienceSharingInRoom(User1, R1.Condition.Result));
        }
}
```

The logic can be refined in our context application with new rules that provide logic that the services are available. The next level rule will refine this reasoning by for example using the following logic:

```
/*ESR Peson ambience sharing in Room
        R1: User has requested privacy in the room (check using Amigo UMPS service).
        ELSE R2: MediaRendering and MediaCapture services are discovered in the Room
        ELSE R3: MediaRendering services are discovered in the Room
    ELSE R4: MediaCapture services are discovered in the Room
    DEFAULT No available A/V services */
```

This logic of high level adaptive composition logic can now be invoked by Java commands:

```
RobertoAmbience = new PersonAmbienceSharing("TestPersons.owl#Roberto");
RobertoAmbience.Trigger();
MariaAmbience = new PersonAmbienceSharing("TestPersons.owl#Roberto");
MariaAmbience.Trigger();
```

The execution of resulting composition depends on the selected composition strategy. One such strategy will be to use rules to select a suitable abstract service for the situation leaving the details of composition and execution to SD-SDCAE, or it can simply be the execution of a user-defined agent programmed in Java that is selected by the rules and uses the selected services to publish a new composed service.

The trace of composition logic in the selected scenario can be examined in VantagePoint to verify that it works as expected against the set of Amigo-S services that are available in the current room.

## 5.4 How to develop an application that integrates context-aware services

The current version of the CASD service is based on basic service discovery (WS-Discovery). The following section will describe the run-time behaviour and examples of the context descriptions of the service and the client. For integration with the semantic discovery a number of subjects should be addressed:

- The service description as passed to CASD should be changed to a semantic description as is used for semantic service discovery.
- The underlying service discovery mechanism should be changed from something like WS-Discovery to the semantic discovery.
- The manner in which context is linked (dynamically) to a service, currently done with a property pointing to a context source, should be changed to be compatible with the semantic service discovery. Optionally this can be done when the service registers with the semantic service repository by including the link to the context source in the semantic service description. Additionally the types of context that are supported y this context source can be included in the description. Having the context itself included in the service description is inefficient, since the very nature of context is dynamic and would therefore lead to a lot of service description updates (whenever the context changes).

### CASD service interface

The CASD service has the interface shown below in code.

The lookup method is used for the request/response style of service discovery (i.e. lookup services taking context of this moment into account, but do not provide further updates).

The doPersistentLookup and unsubscribePersistentLookup methods are used for persistent context aware service discovery, so in case a discovery client wants to be notified whenever a ‘better’ service becomes available based on the context aware matching done by CASD.

```
public interface IAmigoCASD {
    /* Active Service Discovery */
    public String[] lookup (String requiredServiceDescription, String[] contextSourceAmigoRefs,
        String casdSelectionIdentifier);

    /* Subscribe for the persistent service discovery */
    public String[] doPersistentLookup(String requiredServiceDescription,
        String[] contextSourceAmigoRefs, String casdSelectionIdentifier,
        String callBackServiceReference);

    /* Unsubscribe from the persistent service discovery */
    public void unsubscribePersistentLookup(String callBackServiceReference);
}
```

The lookup and doPersistentLookup methods return the strings representing the Amigo Service References. With the persistent lookup, the callbackReference specifies the reference of the client that has to be called whenever the result of the lookup changes.

### CASD service client

When a client wishes to make a Lookup or persistentLookup request, then it needs to:

A. Obtain a handle on the CASDAmigoService

B. Describe the service that it is looking for, represented as requiredServiceDescription above.

C. Provide a reference to the client context sources represented as contextSourceAmigoRefs above.

D. Specify which criteria should be used to select the most appropriate service represented by casdSelectionIdentifier, which is the ranking algorithm mentioned earlier. At the moment the following criteria are envisioned: *Closest*, *Fastest*, and *Cheapest*. The client needs to make sure that its context source provides the required information so that casdSelectionIdentifier can be used. E.g. for the Closest service, the client context source should provide client's location as described using the Amigo ontology.

E. If the client is interested in a persistentLookup request, then it also needs to implement callBackServiceInterface and send this reference in the query represented by callBackServiceReference above.

### Client's context description

The Client is the entity which plans to issue a service discovery request towards the CASD service and thus needs to be able to describe its context to perform service selection based on the casdSelectionIdentifier. Within Amigo WP4, there has been extensive work performed on developing a Context-Aware infrastructure that leverages the ontology developed within WP3. It is convenient for the client to utilise the facilities offered by WP4 with respect to describing its context and making it available.

For example: Consider a client that has a certain location and also wishes to optimise its service selection based upon closeness (in terms of GPS coordinates). In this case, the client would describe its location in the longitude, latitude and altitude format using the following RDF fragment (namespaces and data types excluded for the sake of brevity).

```
<?xml version="1.0"?>
<rdf:RDF >
   <UserLocation>
      <probability>0.9</probability>
      <timestamp>2006-10-18T00:00:00</timestamp>
      <hasAbsoluteLocation>
         <AmigoICCS:WGS84Location rdf:ID="location1">
            <longitude>6.8897</longitude>
            <latitude>52.2328</latitude>
            <altitude>0.00</altitude>
         </AmigoICCS:WGS84Location>"
      </hasAbsoluteLocation>
   </UserLocation>
</rdf:RDF >
```

### Service's context description

For the CASD service to perform service selection based on the casdSelectionIdentifier, it also needs to provide the information about its context sources in the service description as well as the context sources should provide necessary context information so that a query based on the casdSelectionIdentifier can be executed, this is currently done by setting a property to point to the CS.

An example of the ambulance location context source has already been given in Chapter 4.

Services need to publish a reference to their ContextSource when they advertise themselves. For the WS-Discovery case this is done by adding a property ContextSourceURL to the Amigo service description; in the OSGi deployment framework this can be done as follows:

```
ambulanceService.addProperty("ContextSourceURL",serviceContextSource.getReference().getUrl());
```

Where serviceContextSource is the context source providing context about this particular service.

### Obtain a handle on the Context Aware Service Discovery service

The serviceType of CASD service in the Amigo Framework is CASDService. The CRA executes the following (pseudo-code) function to obtain a handle on the CASDService:

```
private AmigoService FindCASDService() {
        // AmigoLdapLookup lookup is the normal discovery service
        CASDService = lookup.lookupFirstService("urn:amigo", "CASDService");
        if (CASDService == null) {
                return null;
        } else {
                return CASDService;
        }
}
```

### Reference to the CRA context source

In the final CRA prototype, the CRA service will use the context source, the context information of which will trigger the rule at ANS. However, CRA is not yet integrated with ANS. Hence, we create a location context source for CRA and provide this information to the CASD service.

The CRA Context Source which provides location is activated as follows. Please note that some steps are omitted for brevity. Refer to the CMS tutorial to see how to develop Context Sources for the Amigo framework.

```
ContextSourceManager manager;
LocationContextSource locationcs = new LocationContextSource(manager);
locationcs.init();
```

The reference to the locationContextSource is obtained as follows:

```
String[] contextSourceRefs = new String[] { locationcs.getAmigoReference().toString() }
```

### Specification of the casdSelectionIdentifier

CRA is currently interested to find the closest ambulance service. Hence it uses the following casdSelectionIdentifier:

```
String selectionIdentifier = "CLOSEST";
```

### Invoking the CASD service

Once the above parameters are specified, the CASDlookup function is invoked as follows:

```
GenericStub stub = SSMDservice.getGenericStub();
String[] argNames = new String[]{"requiredServiceDescription", "contextSources", "casdSelectionIdentifier"};

Object[] argValues = new Object[]{desc, refs, selectionIdentifier};
Object response = stub.invoke("lookup", argNames, argValues);
```

Based on this information, the CASD service will return the CLOSEST ambulance service.

### Interpreting the CASD service response

After obtaining the response from the CASD service, one needs to obtain the Amigo reference of the returned service(s). How this can be done is show using the following pseudo-code:

```
Vector references = (Vector) response;
String refString = (String)references.elementAt(0);
AmigoReference ref = new AmigoReference(refString);
```

This Amigo Reference to the service is to be used to invoke the service.

### Using persistent lookup

To use the CASD Persistent Lookup, the CRA implements the following interface:

```
public interface ICASDCallBack {
        public void notifyServiceChange(String[] newServiceRef);
}
```

The reference to the callBackInterfaceImpl is obtained as follows:

```
String callBackServiceRef = new String (this.callBack.getAmigoReference().toString());
```

The persistentLookup is invoked as follows:

```
String[] argNames = new String[]{"requiredServiceDescription", "contextSourceAmigoRefs",
        "casdSelectionIdentifier", "callBackServiceRef"};
Object[] argValues = new Object[]{requiredERServiceDescription, contextSourceAmigoRefs,
        casdSelectionIdentifier, callBackServiceRef};
Object response = stub.invoke("doPersistentLookup", argNames, argValues);
```

The request to unsubscribe from the Persistent Lookup is invoked as follows:

```
String[] argNames = new String[]{"callBackServiceReference"};
Object[] argValues = new Object[]{callBackServiceReference};
Object response = stub.invoke("unsubscribePersistentLookup", argNames, argValues);
```

### Specification of the service context source

Currently the AmbulanceService is the only implemented Emergency Response Service. The AmbulanceService Context Source, which provides its location, is activated similar to the CRA Context Source as follows. Please note that some steps are omitted for brevity.

```
ContextSourceManager manager;
LocationContextSource locationcs = new LocationContextSource(manager);
locationcs.init();
```

The CotnextSourceURL property is published as follows:

```
AmbulanceService.addProperty("ContextSourceURL", locationcs.getReference().getUrl());
```

## 5.5 How to develop an application that integrates QoS-aware services

In this section, the steps that take place in order to instantiate an application that integrates QoS-aware services are described. A client, herein called a User, submits a request to the Dispatcher, which was sketched in Section 5.1.2.6 for our future integration of approaches. Subsequently, the Dispatcher initializes the QoS-aware Service Selection Tool (QASST) accompanied with the User Task (i.e., task), which describes the required functional capabilities of the requested service set by the User, and the identifier of the Dispatcher itself. QASST uses this User Task object and the SD-SDCAE mechanisms in order to obtain a list of services from the service repository. Thus, QASST obtains a WSDL file from the Service Repository describing the functional and non-functional capabilities of the services that match the submitted request. Having obtained this list of services, QASST applies a selection algorithm and identifies the most appropriate service for this request. In order for the QASST to function properly, the specific QoS preferences of the User for this service type have to be

obtained (service type is described in the WSDL document and declares what is the type of the service, for example the VideoDelivery service described in Chapter 4). Initially, it communicates with the Context Management Service (CMS) component and gets the identification of the User that submitted the request and afterwards it communicates with the User Modelling and Profiling Service (UMPS) component (for CMS and UMPS, see the Amigo [D4.X] deliverables) and obtains the QoS preferences of this User for this service type. Then, a semantic matching takes place and filters out the services that do not address the QoS preferences of the User. Subsequently, a selection algorithm is applied and a single service is selected as the most appropriate one with regards to the specific QoS preferences of the User and the provided QoS capabilities of the services. A reference of this service is passed then to the Dispatcher (through the Dispatcher identifier). The following diagram depicts the steps that have just been described. In case the requested application requires the integration of more than one QoS-aware services, the process that takes place is similar. The QASST matches sequentially the QoS properties required with the QoS properties provided and identifies the most appropriate services that need to be composed to provide an integrated QoS-aware application.

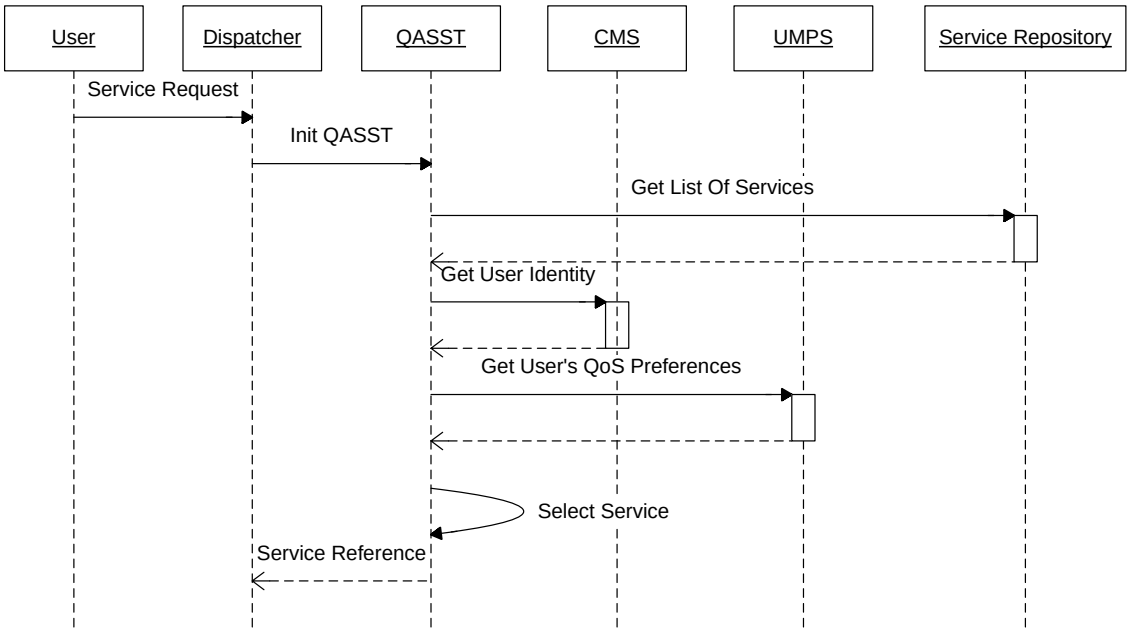


*Figure 5-9: QoS-aware service selection process*

## 5.6 How to develop an application that integrates event-based services

An event-based application is written using a scripting language, named Pantachou, which integrates ontology elements such as semantic services with their parameters. Therefore, a semantic description must be provided. A guideline to provide such a description is available in section 4.2.4.

The use of the Pantachou language to develop an event-based application is described through the LightManager example shown in Figure 5-10. First, the name of the service is defined as a URI (Line 1). The semantic service (*LightManager, Line 1*) defines the type of the Pantachou program and introduces constraints on the service. Then, required services are declared as in the *lumSender* statement (Lines 3 to 9). This statement defines the profile of these services, namely the *LuminosityEventOuputProfile* profile. Besides, the profile is refined

by specifying service properties such as its location. At runtime, the *lumSender* statement will be instantiated as light sensors by the service discovery process.

A service may send several types of events, involving several profiles. To select the right event, the event declaration (line 11) specifies the expected type of values (e.g., *Luminosity* from *lumSender* services).

The receptionBehavior declaration (line 13) defines operations performed when a unique kind of event is received (line 14). The *initial* statement (line 25) initializes the service. In Figure 5-10, the previously defined *lumEvtBehavior* behaviour is selected via the *adopt* statement.

Note that you can attach another behaviour to a given event of the same kind during the life cycle of the service. The previous behaviour of the event will be automatically released. Thus, a single behaviour is attached to a given event. The behaviour concept allows the dynamic reconfiguration of Pantachou services.

The service discovery is performed at runtime. Besides, discovered services are regularly updated towards the expected profile and properties. So, new services are taken into account and unavailable services are not considered any more.

For example, when new light sensors are deployed, the light manager automatically receives events from them. Likewise, when an on/off command is performed on lights, lights registered after the *LightManager* service deployment, also receive the command. Furthermore, the resulting Pantachou application only exposes the functionalities it requires, thus improving its portability, including making it forward compatible with future or refined versions of the requested services (e.g., lights and light sensors).

```
1   'http://ws.amigo.eu/light_manager' instantiates LightManager {
2
3     service Light light {
4       location = hall;
5     };
6     service with output event<Luminosity> lumSender {
7       location = outdoor;
8       unit = binary; // the event will be able to take only two values
9     };
10
11    event<Luminosity> lumEvt from lumSender<*> { };
12
13    receptionBehavior lumEvtBehavior {
14      void receive(lumEvt e) {
15        if (e.value == "NIGHT") {
16          light<*>.on();
17        } else {
18          light<*>.off();
19        }
20      }
21    };
22
23    // ------- Main Section --------
24
25    initial {
26      adopt(lumEvtBehavior);
27    }
28  }
```

*Figure 5-10: Pantachou example of the LightManager*

## 5.7 Resources

[D3.3] Amigo Consortium. Deliverable D3.3: Amigo Middleware Core Enhanced: Prototype Implementation & Documentation. October 2006. Available at: http://www.hitech-projects.com/euprojects/amigo/index.htm.

[D4.X] The D4.X deliverables are available at the Amigo public website: http://www.hitech-projects.com/euprojects/amigo/index.htm.

# 6 How to develop a domotic service

## 6.1 Overview

### 6.1.1 Objectives

There are a number of off-the-shelf domotic systems and devices that can be installed at home but that can't be directly integrated within the Amigo platform.

Therefore, the current document will describe the steps required to develop a domotic service, integrating a domotic device into the Amigo system, by means of Microsoft C# code snippets.

Besides its particularity, a domotic service is a basic Amigo service and employs the Amigo .Net programming and deployment framework (see Chapter 2).

### 6.1.2 Principles and features

The Amigo Domotic Infrastructure aims at presenting heterogeneous physical hardware devices as unified software services using standard service technologies. Nowadays, there is a great diversity of physical device technologies and protocols. Further, there are a number of service technologies that should be supported within the Amigo system.

Therefore, the purpose of the Amigo Domotic Infrastructure is to enable the integration of different device technologies presenting them by means of software services, but isolating the final users (service clients) from the specific base technologies.

Figure 6-1 depicts the proposed architecture. The architecture is based on extracting the required information about the physical devices by means of drivers to the base technologies (BDF, EIB, X10…); modelling the services using a well-known domotic service specification; and building proxies for the domotic model instances using standard service technologies (UPnP, Web Services…).

The intermediate domotic instances decouple the low level drivers from the high level drivers.

Interoperability is achieved by providing several service infrastructures (UPnP, Web Services…) simultaneously to access the same device: for instance, a washing-machine or a lamp can be discovered and controlled either using UPnP or a Web Service (WS-Discovery).

The following components will be developed:

- Domotic Service Model Specification
- Low Level Drivers
- High Level Drivers

### 6.1.3 Assessment

The Amigo Domotic Infrastructure provides an extensible solution for incorporating the functionalities offered by domotic devices into the Amigo system. Devices belonging to different domotic technologies are abstracted as software services, isolating the final users (service clients) from the specific base technologies. The main advantages of using this feature are simplicity for application developers and extensibility for middleware developers. More specifically, application developers are isolated from low-level issues like base technologies, domotic networks, protocols and communications, and only deal with well-known service infrastructures like UPnP and Web Services.

Further, extensibility is assured in two ways. First, new domotic networks and protocols can be incorporated into the current system, by just adding a new low-level driver that will benefit from

the existence of high-level drivers to publish the new services. And second, hitherto unsupported service infrastructures can also be incorporated, by just implementing a new high-level driver that will benefit from the existing low-level drivers to get access to base technologies.

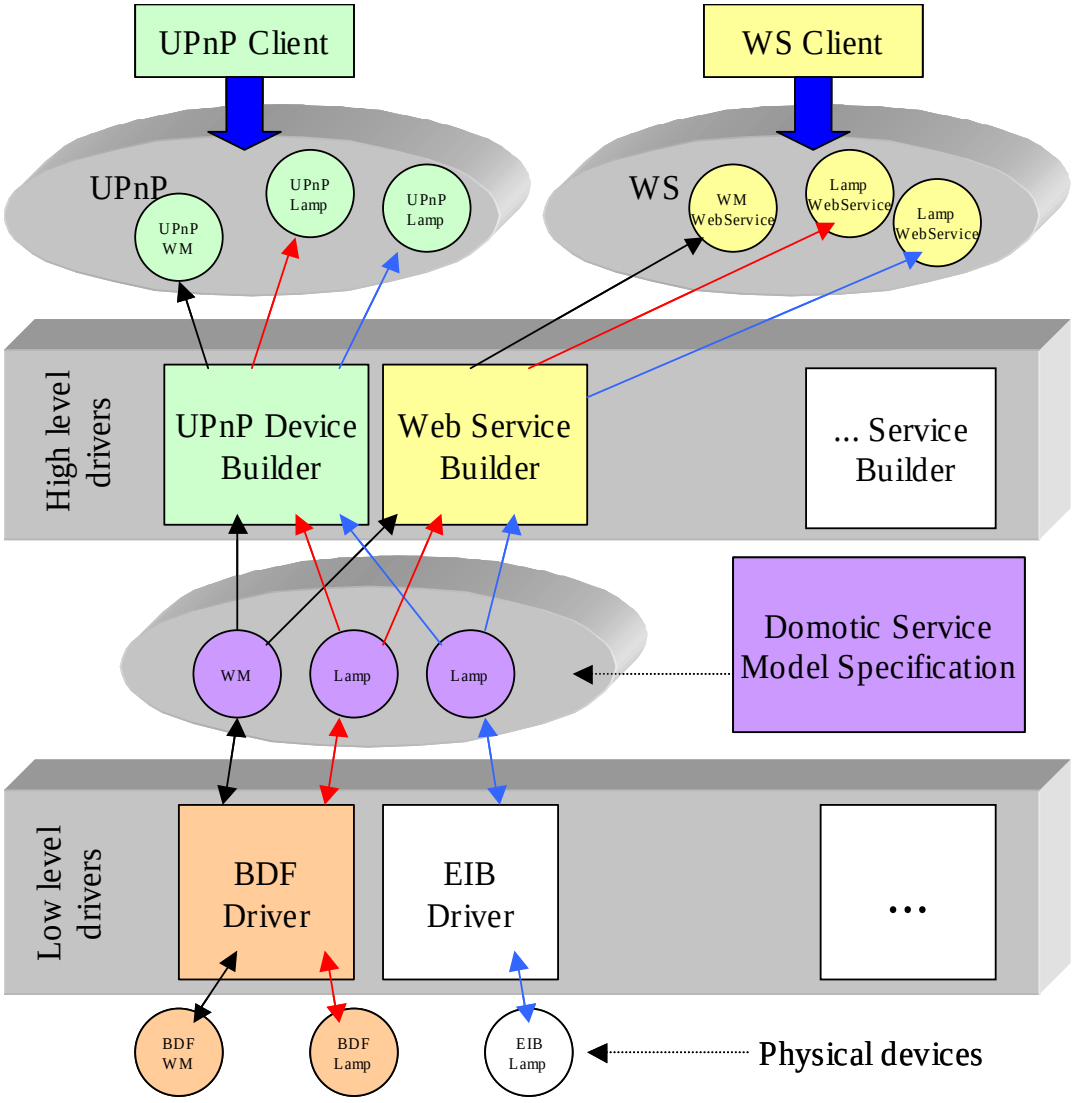


*Figure 6-1: The Amigo Domotic Infrastructure*

## 6.2 Motivating example

As previously mentioned, a set of domotic devices can be easily purchased. But two main difficulties will arise when trying to integrate them with our Amigo home system:

- Proprietary protocols: they will not be able to “speak” with each other.
- Not directly supported by Amigo system (none of them will be accessible by means of service technologies supported by Amigo).

For instance, Amigo users decide to install at home a domotic lamp, a proximity sensor and a domotic washing-machine. The three of them use proprietary, not-interoperable protocols, so trying to use them together is not an easy task.

The three manufacturers provide us with a Software Development Kit (SDK) for their own device, but unfortunately they are absolutely different: the sensor is accessed using a proprietary protocol over RS232, the washing-machine is based on ActiveX components and the lamp is based on some .NET classes.

The Amigo Domotic Infrastructure will allow users to use the three devices, discovering and interacting with software services: Web Services or UPnP devices.

## 6.3 Understanding the domotic service model

In order to integrate heterogeneous domotic devices, an abstract description of the available services, not attached to specific domotic technologies, must be specified. This intermediate description is the common element in the domotic proxy generation process. This component provides any domotic service developer with the abstract reference of the service description.

Any domotic device can be modelled as a provider of a set of services:

Conceptually, any lamp offers a “lighting” service that consists of a state variable that describes the state of the lamp (on/off), a method to set this state, a method to retrieve it and, probably, an event that notifies the change of the state. In a similar way, a proximity sensor can be seen as a service that raises an event when movement is detected nearby. Finally, a washing-machine could be described as a set of properties (status, program, temperature, level of water, start time…), a set of methods (on, off, change program, pause…) and some events (program is finished, water leak alarm…).

Depending on the device capabilities and complexity, it will require more or less methods, properties and events, but the functionality of any device should be capable of being described using these terms.

Further, any device could have non functional properties like owner, location, authorization policies… that can be modelled as metadata as well.

Therefore, a number of interfaces are defined to model any domotic device. Figure 6-2 shows these interfaces and some of their relations. For the sake of simplicity, only a subset of members has been depicted.

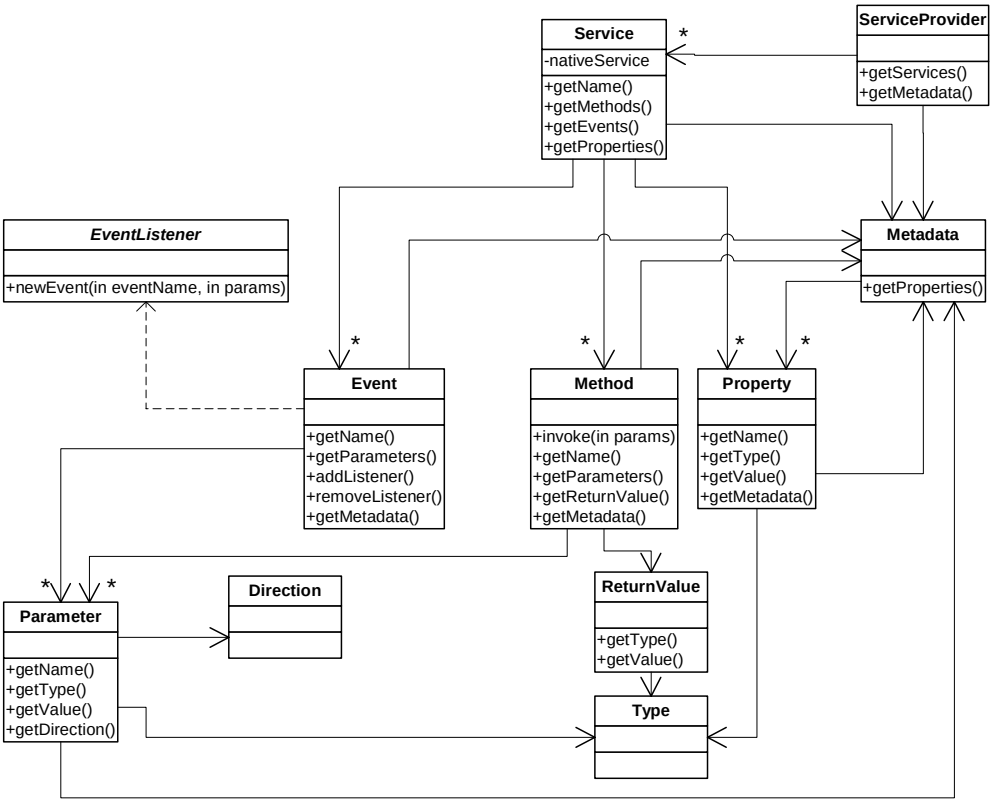


*Figure 6-2: Domotic service model interfaces*

The interfaces are listed below:

- Service Provider: any device can be modelled as a Service Provider that has a name, some Metadata and offers some services.

```
public interface IServiceProvider
{
        string getName();
        Hashtable getServices();
        IMetadata getMetadata();
        bool addService(IService objService);
        void setMetadata(IMetadata objMetadata);
}
```

- Metadata: consists of a group of properties.

```
public interface IMetadata
{
        Hashtable getProperties();
        bool addProperty(Property objProperty);
}
```

- Service: has a name, some methods, some properties and some events.

```
public interface IService

{
        string getName();
        Hashtable getMethods();
        IMethod getServiceMethod(string methodName);
        IProperty getServiceProperty(string propertyName);
        IEvent getServiceEvent(string eventName);
```

```
        Hashtable getEvents();
        Hashtable getProperties();
        IMetadata getMetadata();
        bool addMethod (IMethod objMethod);
        bool addEvent (IEvent objEvent);
        bool addProperty(IProperty objProperty);
        void setMetadata(IMetadata objMetadata);
}
```

- Method: is defined by a name, some parameters, a return value and metadata.

```
public interface IMethod
{
        IMetadata getMetadata();
        IReturnValue getReturnValue();
        ArrayList getParameters();
        IReturnValue invoke(ArrayList parameters);
        void setObject(object obj);
        void setMethodInfo(System.Reflection.MethodInfo methInfo);
        void setMetadata(IMetadata objMetadata);
        void setReturnValue(IReturnValue objReturnValue);
        bool addParameter(IParameter objParameter);
        string getName();
}
```

- ReturnValue: has a type and a value.

```
public interface IReturnValue
{
        System.Type getReturnValueType();
        void setType(System.Type objType);
        object getValue();
        void setValue(Object objValue);
}
```

- Parameter: is described by its name, value, direction, type and metadata.

```
public interface IParameter
{
        string getName();
        void setName(string strName);
        System.Type getParameterType();
        void setType(System.Type objType);
        object getValue();
        Direction getDirection();
        void setDirection(Direction direction);
        void setValue(Object objValue);
}
```

- Direction

```
public enum IDirection
{
        IN, OUT
}
```

- Property: has a name, value, type and metadata.

```
public interface IProperty
{
        string getName();
        object getValue();
        System.Type getPropertyType();
        IMetadata getMetadata();
        void setMetadata(IMetadata objMetadata);
        void setValue(Object objValue);
        void setType(System.Type objType);
        void notifyListeners();
        void setObject(object obj);
        void setPropertyInfo(System.Reflection.PropertyInfo propInfo);
        void eventHandler(ArrayList parameters);
}
```

- Event: described by its name, parameters and metadata.

```
public interface IEvent
{
        string getName();
        ArrayList getParameters();
        IMetadata getMetadata();
        bool addListener(IEventListener listener);
        bool removeListener(IEventListener listener);
        void setMetadata(IMetadata objMetadata);
        void notifyListeners(ArrayList parameters);
        bool addParameter(IParameter objParameter);
        void eventHandler(ArrayList parameters);
}
```

- EventListener: defines the interface that event listeners must implement.

```
public interface IEventListener
{
        void newEvent(string strEventName, ArrayList parameters);
}
```

### 6.3.1 Use example

The following example shows how to model a lamp, instantiating a service provider. The manufacturer offers a .NET class to interact with the lamp: LampManufacturer.clsLamp

This class offers:

- Two methods (GetStatus and SendStatus)

```
public bool GetStatus()
public void SetStatus(bool blnStatus)
```

- An event (NewStatus)

```
public delegate void StatusChangedDelegate(bool blnNewStatus);
public event StatusChangedDelegate NewStatus;
```

An object of this class could be instantiated, granting the access to the physical lamp.

```
LampManufacturer.clsLamp legacyLamp = new LampManufacturer.clsLamp();
```

Considering that:

- IUSM (Interfaces Unified Service Model) defines a namespace for the interfaces
- USM (Unified Service Model) defines a namespace for a set of classes that implement those interfaces

#### ServiceProvider

The physical lamp (located in the bedroom) can be modelled as a ServiceProvider called “BedroomLamp”.

```
USM.ServiceProvider myLamp = new USM.ServiceProvider("BedroomLamp");
```

#### Service

The lamp will offer a service (lighting service), so we should instantiate a Service called “LightingService”:

```
USM.Service myLightingService = new USM.Service("LightingService");
```

This service is provided by myLamp so it should be assigned properly:

```
myLamp.addService(myLightingService);
```

### Method

The GetStatus method should be defined in a similar way:

```
USM.Method GetStatusMethod = new USM.Method("GetStatus");
```

GetStatus has a return value, so it should be defined and assigned to the method:

```
USM.ReturnValue GetStatusMethodReturnValue = new USM.ReturnValue ();
GetStatusMethod.setReturnValue(GetStatusMethodReturnValue);
```

This method should be linked with the native method (GetStatus) so that invoking the modelled method should invoke the native one. This could be implemented using Reflection techniques.

```
GetStatusMethod.setMethodInfo(
        legacyLamp.GetType.GetMethod("GetStatus"));
GetStatusMethod.setObject(legacyLamp);
```

Now, the modelled method has a reference to the native one.

Finally the method should be assigned to the service:

```
myLightingService.addMethod(GetStatusMethod);
```

The SetStatus method should also be defined:

```
USM.Method SetStatusMethod = new USM.Method("SetStatus");
```

The SetStatus method has a parameter that must be defined and assigned to the method:

```
USM.Parameter SetStatusParameter = new USM.Parameter("blnStatus");
SetStatusParameter.setDirection(IUSM.Direction.IN);
SetStatusParameter.setType(typeof(bool));
SetStatusMethod.addParameter(SetStatusParameter);
```

And finally the method should be assigned to the service:

```
myLightingService.addMethod(SetStatusMethod);
```

### Event

Events follow the same principles. First it must be defined:

```
USM.Event StatusChangedEvent = new USM.Event("StatusChanged");
```

Then a parameter must be defined and assigned:

```
USM.Parameter NewStatusParameter = new USM.Parameter ("NewStatus");
NewStatusParameter.setDirection(IUSM.Direction.IN);
NewStatusParameter.setType (typeof(bool));
StatusChangedEvent.addParameter (NewStatusParameter);
```

Modelled events should also be linked to native ones using Reflection techniques, and finally the event should be assigned to the service:

```
myLightingService.addEvent(StatusChangedEvent);
```

### Metadata

Additionally, Metadata can be added to any of the previously described elements.

For instance, the location of the lamp can be expressed in the following way:

```
USM.Metadata LampMetadata = myLamp.getMetadata ();

USM.Property LampLocationProperty = new USM.Property ("Location");
```

```
LampLocationProperty.setType (typeof(string));
LampLocationProperty.setValue("Bedroom");


LampMetadata.addProperty (LampLocationProperty);
```

And a method could also be annotated:

```
USM.Metadata SetStatusMetadata = SetStatusMethod.getMetadata ();
USM.Property AuthorizationProperty = new USM.Property ("Authorization");
…
```

## 6.4 How to develop a low-level driver

Low Level drivers are the components responsible for instantiating ServiceProviders. They act as ServiceProvider factories and, obviously, they are base technology dependent, because they depend on the legacy technology used to interact with the physical device.

They should follow the steps described in Section 6.3.1 with each base technology. In Section 6.2 three examples have been presented: a proprietary protocol over RS232, ActiveX components and .NET classes, so three dedicated factories are required.

### 6.4.1 Use example

Let's continue with the previous example, considering that instead of a single lamp, we want to install five different lamps using a domotic bus like X10, BDF, EIB, etc. We could, for instance, purchase five off-the-shelf X10 lamps that unfortunately are constrained to X10 proprietary protocols, but not Amigo aware service protocols. So, we require an X10 SDK to be able to "speak" with the X10 bus, exchanging messages with the physical lamps.

Although this SDK is out of the scope of this document, it's enough to know that it should offer mechanisms to interact with the lamps. So, an X10 low-driver, by means of this SDK, should have the responsibility of instantiating as many ServiceProviders as there are lamps in the network, linking the corresponding modelled methods with the interfaces provided by the SDK.

For instance, the X10 SDK could offer the following methods:

```
string[] GetX10Devices()
bool SwitchOn(string DeviceAddress)
bool SwitchOff(string DeviceAddress)
bool AllLightsOn()
…
```

GetX10Devices would return an array of strings containing the addresses of the present lamps ("A1", "A2", "A3", "A4", "A5"); SwitchOn/Off would switch on/off the lamps addressed by DeviceAddress; and AllLightsOn would switch all the lights on.

The X10 driver could, at runtime, interrogate the bus about the lamps using GetX10Devices, and instantiate a ServiceProvider for each lamp.

```
string[]CurrentLampAddresses = X10SDK.GetX10Devices();

foreach (string strLampAddress in CurrentLampAddresses)

{
        USM.ServiceProvider Lamp = BuildLamp(strLampAddress);
        …
}
```

Where BuildLamp could be also defined like:

```
private USM.ServiceProvider BuildLamp(string strAddress)
{
        USM.ServiceProvider TempLamp;
        TempLamp = new USM.ServiceProvider(strAddress);
        /*All required methods, properties, events should be instantiated as described in 6.3.1. */
```

```
        …
        USM.Method OnMethod = new USM.Method("On");
        …
        //The modeled methods should be linked to the native SDK methods
        OnMethod.setObject(X10SDK)
        OnMethod.setMethodInfo(X10SDK.GetType.GetMethod("SwitchOn"))
        …
        return TempLamp;
}
```

## 6.5 How to use a high-level driver

The goal of a high-level driver component is to instantiate High Level proxies (Web Services, or UPnP devices) starting from the generic instances described by the Domotic Service Model component (ServiceProviders).

A high-level driver is absolutely independent from the base technologies because it only requires the information provided by the modelled ServiceProviders.

As shown in Figure 6-1 two high-level drivers are available:

- Web Service builder
- UPnP Device builder

Both of them offer a very simple interface to generate the corresponding proxies.

### 6.5.1 WebService builder

This component uses the .Net Discovery Framework (see Chapter 2), so the instantiated services can be discovered using WS-Discovery: services will automatically advertise themselves on the network when they are instantiated (WS-Discovery: Hello) and announce their leaving (WS-Discovery: Bye) when they are disposed. They also respond to queries (WS-Discovery: Probe and Resolve).

Reflection emit is a run-time feature that allows code to create dynamic assemblies, modules, and types. Instances representing the domotic services, according to the Domotic Service Model component specification, are dynamically created using this feature.

```
DiscoverableService createWS(USM.ServiceProvider serviceProvider)
{
        …
        DiscoverableService miSW = this.createService(service);
        …
        return miSW;
}
```

### 6.5.2 UPnP device builder

This high-level driver instantiates high-level proxies (UPnP proxies) starting from the generic instances described by the Domotic Service Model component.

The proxy instantiation is also a dynamic runtime process. Only a ServiceProvider instance is required to generate the UPnP proxy.

```
UPnPDevice createUPnPProxy(USM.ServiceProvider serviceProvider)
{
        …
        UPnPDevice device = UPnPDevice.CreateRootDevice();
        …
        return device;
}
```

# 7 How to develop a multimedia content application

## 7.1 Overview

In this chapter, the motivation of building an application that uses the Amigo Middleware to manage and render multimedia content, together with the overall methodology to follow, is presented. The following sections are an effort to achieve quick reference documentation for developers and at the same time an overall description of the middleware functionality when dealing with multimedia content. In the examples, Java and C# code samples are provided to illustrate, from a general point of view, how developers may access the described middleware functionalities.

### 7.1.1 Objectives and principles

The Amigo Middleware intends to provide a set of functionalities regarding multimedia content in the home that allows application developers to concentrate on the application itself, and not having to deal with underlying technical problems that arise from the existing technologies and devices related to this domain. At the same time, the middleware should provide functionalities that might lead to innovative applications dealing with multimedia content.

### 7.1.2 Features

The Amigo Middleware provides methods for retrieving and modifying information about UPnP AV devices currently online as well as all of those known by the middleware although currently offline. Some of this information is editable. The middleware provides methods for performing searches, in terms of metadata, through all content present in the home network via UPnP AV servers. Modification of this metadata is also possible. Semantic annotation of content based on OWL can be stored using the middleware, so that semantic information related to given content is available to all applications. Semantic queries can also be executed by the middleware upon the whole semantic knowledge base related to the content at home. Furthermore, methods are provided to select and possibly create resources associated with a given content in formats that are compatible with a given renderer, either automatically by the middleware and thus transparent, or explicitly by the application. Finally, establishing a multimedia rendering session (i.e. playing content on a renderer) and controlling it can also be done through the middleware using only references, without further knowledge of where the content and the renderer actually are.

### 7.1.3 Assessment

The middleware provides transparency to searches, semantic queries and metadata modification when dealing with distributed content. Transparency is provided in the sense that applications deal with a unique interface with no need to maintain dynamic structures representing not always available content items, devices and sessions: discovery of these entities is hidden by the middleware and data structures, persistent and not persistent, are maintained within. In other words, the intrinsically complex state model of devices' discovery, session management and content management in a dynamic environment is reduced by the middleware to a simple RPC model, leading to simple state model or even stateless applications. This enables the application developer to concentrate in the application functionalities themselves rather than on underlying, protocol dependent problems. Furthermore, the existence within the middleware of a semantic knowledge base related to content enables enrichment of content descriptions in a model that also enables reasoning upon it, leading to a wide variety of possible applications. This enrichment is possible using the

OWL import mechanism fully supported by the middleware in a transparent way thus enabling reusability of ontologies external to the Amigo project, which may be richer in a certain knowledge domain, exploiting in this way the reusability principle of ontologies. For instance, a given content semantic description, let's say a film, could be enriched by adding entities defined in an external ontology, let's say an exhaustive knowledge base of film directors. Then an external property (i.e. directedBy) relating the "Film" or "Thing" class to the "Director" class would provide a mechanism for establishing a semantic link within the given film and a complete semantic description of the appropriate director provided by experts in this domain. Furthermore, the middleware provides a method for selecting content items by performing an RDQL query.

## 7.2 Motivating example

Digital entertainment is establishing steadily in our homes. The quantity of content delivered in digital format to the home has risen very significantly over the last ten years. Therefore users are interested in accessing movies, music, pictures… and viewing them in their favourite devices in a seamless way. However, this can not be done nowadays since the user must interact directly with the devices containing the content. An application developer may want new ways of achieving this purpose allowing the homeowner to perform these operations from a variety of UI devices.

For instance, a developer may want to produce an application that enables the user to navigate through the content distributed in different devices acting as servers in a seamless way. He will probably want the user to be able to search content within all the content available at home, not caring where it specifically lies. He would probably also want content inside portable devices to be shown by his application together with their descriptions when brought home by a user, and its contents to disappear from the application navigation screen upon disappearance of the device. It is most probable that a user would like to play content after finding it; so the user being able to select a content item and a renderer, starting a multimedia session and controlling the playback of the item (i.e. play, stop, pause, etc.) seems reasonable. The Media Manager Core application that is under development within the project is an example of this kind of application and it will appear through the following sections as an illustrative example.

## 7.3 Content distribution set up

The Content Distribution functional block of the middleware is made up of four distributed components deeply related. All of them must be running in the same network and, of course, have connectivity with each other. The setup is presented in Figure 7-1.

The Content Distribution Service is the entry-point to all Content Distribution functionalities except generic data storage. The latter is performed by direct dialogue between the application and the Data Store component. The Content Discovery component is in charge of dynamic content discovery, metadata aggregation and semantic functionalities. The Content Adaptation Server provides content transcoding methods based in a plug-in architecture and at the same time is a Content Server that provides storage for content that is desired to be persistent. Specific details on installation and configuration of each component can be found in the corresponding user guides at [AmigoRepository].

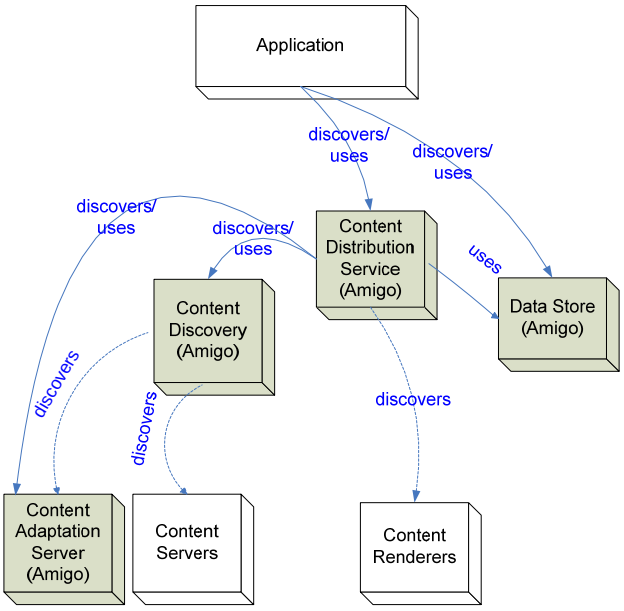


*Figure 7-1: Content Distribution Setup*

## 7.4 Basic principles

As mentioned in the previous section, the Content Distribution Service is the entry point to the system. Therefore, the developer should deal with this component in order to make use of the Content Distribution functionalities. Content Distribution is provided as a Web Service and the model describing it (WSDL) can be found at http://ContentDistributionMachineIP:2343/?wsdl (see Content Distribution guides at [AmigoRepository]). Hence, a developer aiming to use ContentDistribution should create a client to this service. In the following sections, sample Java and C# clients are used to explain how to deal with the ContentDistribution interface. This client could be dynamically generated once the ContentDistribution service has been discovered using WS-Discovery, either via the Java or C# Amigo development frameworks, or any other software that enables SOAP method invocation (.NET Framework native methods, AXIS, etc. ) and, optionally, WS-Discovery discovery.

Moreover, the developer will have to handle DIDL-Lite documents. These documents are used to describe multimedia contents in the connected home by servers compliant with the UPnP AV MediaServer specification [UPnPAVCD]. In order to ease the processing of these documents, the Cyberlink API (see [Cyberlink]) is used together with some classes developed in the present project and bundled in the AmigoCommons.jar (see CADMS or ContentDiscovery javadoc at [AmigoRepository]). Specifically, the ContentNode class from this API represents content described by a DIDL-Lite document and the DIDLLite class provides marshalling and unmarshalling functionalities. Both are used in the samples of code of the sections below.

## 7.5 How to browse content devices

The Amigo Middleware is able to discover all UPnP AV devices (software or hardware based) available at home and organize them in two categories: servers and renderers. Server stands for an UPnP AV MediaServer, a device that provides AV multimedia content to other UPnP devices on the home network. Renderer stands for UPnP AV MediaRenderer, a device capable of rendering AV content from the home network.

In order to discover these devices two main methods are used from the ContentDistribution class.

*getRenderers():* that returns a *GetRenderersResult* object which contains all renderers discovered by the Content Distribution service.

*getContentServers():* that returns a *GetContentServersResult* which contains all servers discovered by the Content Distribution service.

Furthermore it is possible to access all the properties of renderers and servers and modify some of them in order to give the user the capacity of customizing the description of a particular device. The following methods are available:

In order to get the description of a renderer we should invoke the method *getRendererDescriptions()* from the *GetRenderersResult* class which returns an array of *RendererDescription* items. This class offers getters and setters for the following properties of a renderer: *capabilities, friendlyName, manufacturer, manufacturerUrl, modelDescription, modelName, modelNumber, modelUrl, online, rendererId, serialNumber, uniqueDeviceName, universalProductCode, upnpDevice*, and provides an object representation of the result data type for the *getRenderers()* method specified in the WSDL description of the Content Distribution Service. Specifically, this was generated using the *wsdl2java* tool but any other strategy may be adopted. The following code samples in this and next sections presume the usage of this kind of tool for code conciseness.

In the same way, to get the description of a server the method *getContentServerDescriptions()* from the *GetContentServersResult* should be invoked, which returns an array of *ContentServerDescription* items. This class offers getters and setters for the following properties of a server: *contentServerId, friendlyName, iconUrl, manufacturer, manufacturerUrl, modelDescription, modelName, modelNumber, modelUrl*.

An example of how to print the friendlyName of all renderers and servers available is given below

```
// Access to the Content Distribution service (should be discovered)
ContentDistribution service;

// Get rendererDescriptions
RendererDescription[] rendererDescriptions = service.getRenderers().getRendererDescriptions();
// Get serverDescriptions
ContentServerDescription[] serverDescriptions = service.getContentServers().getContentServerDescriptions();

// Print data
for(int i = 0; i < rendererDescriptions.length; i++)
   System.out.println(rendererDescriptions[i].getFriendlyName());

for(int i = 0; i < serverDescriptions.length; i++)
   System.out.println( serverDescriptions[i].getFriendlyName());
```

An example of the MMC showing all the renderers and servers is shown in Figure 7-2.

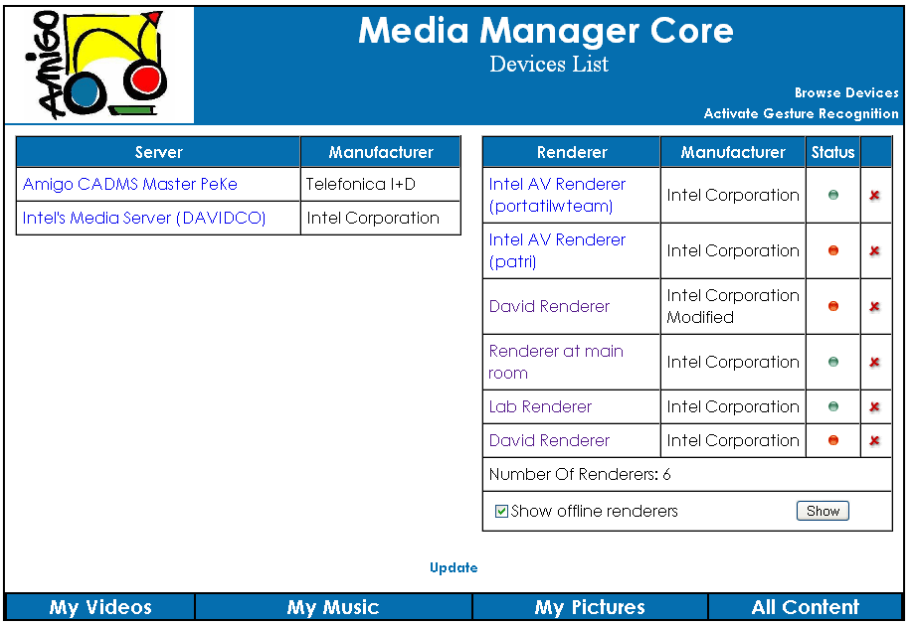


*Figure 7-2: Devices List*

## 7.6 How to find content

The Amigo Middleware allows looking for any content available at home in a seamless way. In order to perform a search successfully the following methods should be used.

First of all a query should be created using the object representation of method parameter types (see previous section).

```
Query query = new Query();
```

Then the query to execute should be specified.

```
query.setQueryString(aQueryString);
```

A query must follow the *SearchCriteria* syntax specified in the ContentDirectory service template available at [UPnPAVCD] or RDQL syntax if the query is addressed to the semantic model of content at the home. Depending on the syntax, Content Distribution will transparently redirect the query to the underlying literal metadata database or semantic knowledge base respectively.

Some examples of literal queries are listed below:

- All items (music, video, pictures, …) available*: upnp:class derivedfrom "object.item"*
- All video items: *upnp:class derivedfrom "object.item.videoItem"*
- All audio items: *upnp:class derivedfrom "object.item.audioItem"*
- All image items: *upnp:class derivedfrom "object.item.imageItem"*
- All movies where Tom Hanks plays: *upnp:actor contains "Tom Hanks"*
- All Shakira songs: *upnp:artist contains "Shakira"*

Afterwards, the query should be executed.

```
FindContentResult contentResult = service.findContent(query);
```

The method *getContentDescriptions()* from the *FindContentResult* class can be used to get all the descriptions of the items returned by the findContent query.

```
ContentDescription[] arrayOfContentDescription =
   contentResult.getContentDescriptions();
```

The *ContentDescription* class has a method called *getDescription()* to get the description of an item as a string following the DIDL-Lite syntax. The *DIDLLite* class should be used to convert this string into a *ContentNode* class that offers several options to manage items.

An example of how to list all the items with its properties available at home is given below:

```
// Access to the Content Distribution service (should be discovered)
ContentDistribution service;

// Search UPNP Criteria
String all = "upnp:class derivedfrom \"object.item\"";

// Create the query
Query query = new Query();

// Set the query
query.setQueryString(all);

// Get serverDescriptions
ContentServerDescription[] serverDescriptions =
   service.getContentServers().getContentServerDescriptions();

// Execute the query
FindContentResult contentResult = service.findContent(query);

// Get the Descriptions of Contents
ContentDescription[] contentDescriptions = contentResult.getContentDescriptions();

// Iterate through each Content Description
for (int i = 0; i < contentDescriptions.length; i++){
  // Get Description for Content at position i (DIDL Lite string file)
   String strDIDLLite = contentDescriptions[i].getDescription();

   // Replace < and > characters in DIDL Lite string
   strDIDLLite = strDIDLLite.replaceAll("&lt;", "<");
   strDIDLLite = strDIDLLite.replaceAll("&gt;", ">");

   // Create a DIDL Lite for Content description
   DIDLLite didl = new DIDLLite();

   // Get the Content Node from the DIDL Lite
   ContentNode contentNode = didl.toContentNode(strDIDLLite);

   for (int n = 0; n < contentNode.getNProperties(); n++){
      Property prop = contentNode.getProperty(n);
      System.out.println("Property " + prop.getName() + ": " + prop.getValue());
   }
}
```

An example of the MMC showing all the content available is shown in Figure 7-3.

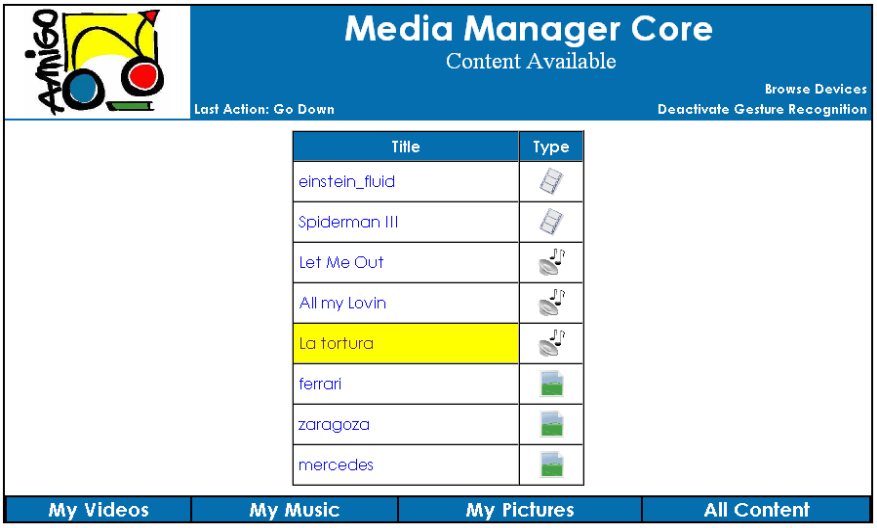


*Figure 7-3: MMC Content Available screen*

## 7.7 How to update content metadata

Each item has some content metadata associated with it. This content metadata can be modified using the following methods:

```
// Access to the Content Distribution service (should be discovered)
ContentDistribution service;

// Content Node that we want to update
ContentNode contentNode;

// Update a property of the content Node
contentNode.setProperty(propertyName, propertyValue);

// Create a ContentNode list
ContentNodeList contentNodeList = new ContentNodeList();

// Add the ContentNode to the list
contentNodeList.add(contentNode);
DIDLLite didlLite = new DIDLLite();

// Convert the ContentNode to a DIDL Lite document
String didlDescription = didlLite.toString(contentNodeList);

// Create a ContentDescription from the ContentNode
ContentDescription cd =
    new ContentDescription(new ContentId(contentNode.getID()),didlDescription);

// Update the content metadata
service.updateContentDescription(cd.getContentId(), cd);
```

Many different kind of properties could be modified. These properties can be found in ContentDirectory template available at [UPnPAVCD] . Some examples of are listed below:

- Title: "dc:title"
- Class: "upnp:class"
- Actor: "upnp:actor"
- Date: "dc:date"

An example of the application MMC performing this action is shown in Figure 7-4:

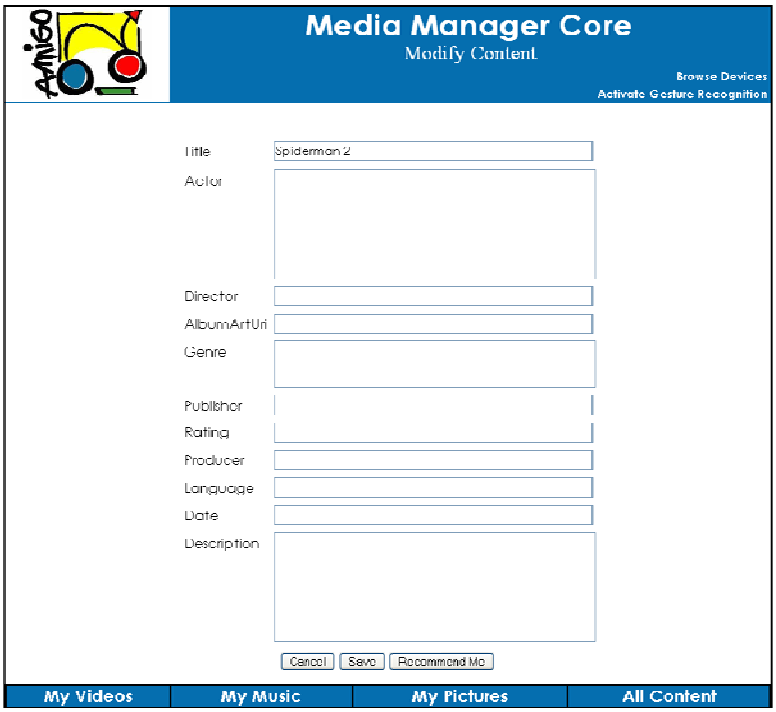


*Figure 7-4: MMC Modify Content*

Semantic information is embedded in the DIDL-Lite description of multimedia items under a <*desc*> element with the same *id* attribute as the item and *nameSpace* value of *urn:amigo-schemas:multimediaContent*. Inside the <*desc*> element a single element <*semantic*> from this namespace contains the semantic description as CDATA. The iterative embeddings are there to keep a complete and valid DIDL-Lite instance (see [UPnPAVCD]). The following is an example of semantic information embedded in the DIDL-Lite:

```
<item id="7" parentID="2" restricted="0">
   <dc:title>Sunflower</dc:title>
   <upnp:class>object.item.audioItem.musicTrack</upnp:class>
   <upnp:artist>Paul Weller</upnp:artist>
   <desc id="7" nameSpace="urn:amigo-schemas:multimediaContent">
    <semantic xmlns="urn:amigo-schemas:multimediaContent">
     <![CDATA[
      <rdf:RDF
        xmlns:amigo="http://www.owl-ontologies.com/Amigo/Amigo.owl#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl=http://www.w3.org/2002/07/owl#
        …
        …>
        <owl:Ontology rdf:about="http://www.owl.org/Amigo/MultimediaContent.owl">
          <owl:imports rdf:resource="http://www.owl.org /Amigo/Amigo.owl"/>
          <owl:imports rdf:resource="http://www.owl.org/Amigo/Multimedia.owl"/>
        </owl:Ontology>
          …
          …
      </rdf:RDF>
     ]]>
    </semantic>
   </desc>
</item>
```

The URL's and semantic information inside the CDATA in the previous example are not syntactically correct. However, it conveys the general idea. In a real application the data inside CDATA must be a consistent OWL document. For detailed information see [OWL] and the latest version of the Amigo Multimedia Content Ontology [MultimediaOWL]. Applications can modify this information, enriching it with external public ontologies, and save it as the rest of the metadata is done. However, usage of the Amigo Multimedia Content Ontology individuals as a base is mandatory: the application is only free to relate these to external concepts using external properties, but must not remove these individuals or substitute them by different ones. This must be done using the appropriate content distribution service method for removal of content items.

In order to modify semantic metadata, the code would be similar to the one above.

```
// Access to the Content Distribution service (should be discovered)
ContentDistribution service;

// Content Node that we want to update
ContentNode contentNode;

// Semantic info is a string containing <semantic xmlns="urn:amigo-
// schemas:multimediaContent"> … </semantic>

String semanticInfo;

//… Some code here completing the semanticInfo …

Property property= new Property("desc",semanticInfo);
property.addAttribute("id", cnode.getStringID());
property.addAttribute("namespace", "urn:amigo-schemas:multimediaContent" );

// Update a property of the content Node
contentNode.setProperty(property);

// Create a ContentNode list
ContentNodeList contentNodeList = new ContentNodeList();
// Add the ContentNode to the list
contentNodeList.add(contentNode);
DIDLLite didlLite = new DIDLLite();
// Convert the ContentNode to a DIDL Lite document
String didlDescription = didlLite.toString(contentNodeList);

// Create a ContentDescription from the ContentNode
ContentDescription cd = new ContentDescription(new ContentId(contentNode.getID()), didlDescription);

// Update the content metadata
service.updateContentDescription(cd.getContentId(), cd);
```

## 7.8 How to start and control playback sessions

The Amigo Middleware offers the functionality of playing a selected item resource on a device. A general description of how to achieve it is given below.

```
// Access to the Content Distribution service (should be discovered)
ContentDistribution service;

// Renderer id of the renderer where we want to play the data
RendererId rendererId

// Content id of the content that we want to play
ContentId contentId;

// Get the resources associated to a content
ResourceDescription[] resourceDescriptions =
   service.getResources(contentId).getResourceDescriptions();

// Get the resource id of the resource that we want to play
```

```
ResourceId resourceId = resourceDescriptions[0].getResourceId();

// Play the content on the renderer
service.play(resourceId, rendererId);

// Retrieve session descriptions and pause all of them
SessionDescription[] sdescriptions = service.getSessions();
for (i=0; i< sdescriptions.length; i++)
   service.pause(sdescriptions[i].getSessionId());
```

Then by invoking the *getSessions()* method the ongoing media sessions are retrieved. Using their ids, an application is able to control them. An example of the C# code performing pausing all ongoing media sessions is provided below (above is the example in Java):

```
contentDistributionProxy.GetSessionsResult getSessionsResult;
getSessionsResult = contentDistributionProxy.GetSessions();

if (getSessionsResult.Success)
{
   foreach (contentDistributionProxy.SessionDescription sessionDescription
         in getSessionsResult.SessionDescriptions)
   {
// Simply pause all ongoing sessions
   contentDistributionProxy.Pause(sessionDescription.SessionId);
   }
}
else
{
   Console.WriteLine("Error: Gettting session list failed" +
     " with the following message: " + getSessionsResult.Reason);
}
```

An example of the MMC playing content is given in Figure 7-5:

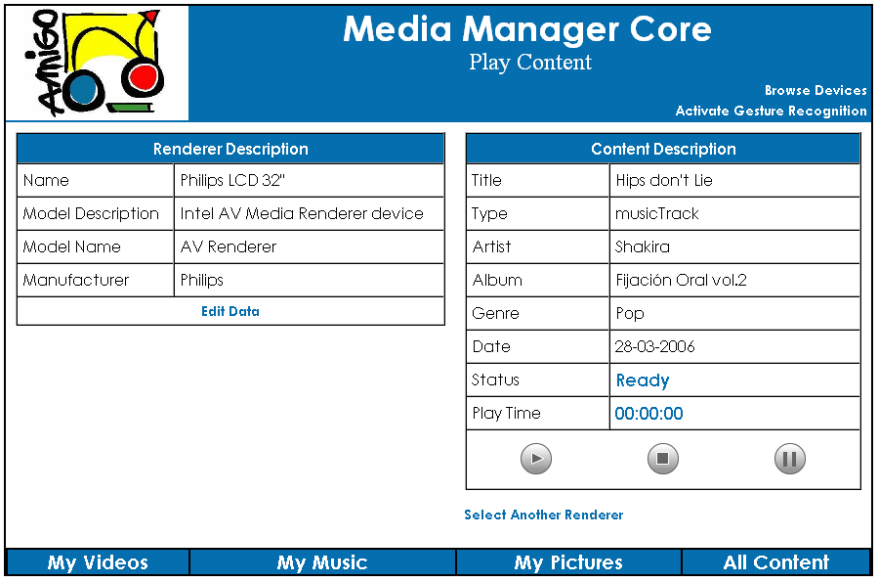


*Figure 7-5: MMC Play Content*

## 7.9 How to adapt and use adapted content

To support different hardware needs within the Amigo network, the Content Distribution service supports an automatic adaptation of content to the needs of any renderer that is known to the service. You can simply add a renderer description to the service which contains some metadata and the device capabilities description as a XML rendered CC/PP profile. After a certain amount of time that is needed for the adaptation process, adapted content will be available that fits the need of any renderer known to the service.

The following example in C# shows how those two steps are accomplished. The first step shows how a renderer is made visible to the system.

```
contentDistributionProxy.AddRendererResult addRendererResult;
contentDistributionProxy.RendererDescription rendererDescription =
   new contentDistributionProxy.RendererDescription();

rendererDescription.ModelDescription = "Example Renderer";
rendererDescription.UniqueDeviceName = "{68FF7123-4087-4ed6-A856-F7ADB211C4A9}";
rendererDescription.Capabilities = "…";
addRendererResult = proxy.AddRenderer(rendererDescription);
```

A new renderer is added manually to the system. Additionally to the already known renderers, that can be found using SSDP and are automatically found by the service, there might be some renderers that have to be added this way. A model description is assigned, as well as the mandatory and unique device name, which is a GUID in our case. A string is assigned to the capabilities member which contains the CC/PP profile. The specification of the CC/PP profile is beyond the scope of this document; please see the References section for further details.

After adding this renderer, the system will start to adapt the content for this specific device. The adapted content can now be accessed like shown in the following example:

```
contentDistributionProxy.GetMatchingResourceResult
getMatchingResourceResult;
getMatchingResourceResult = contentDistributionProxy.GetMatchingResources
(contentID, rendererId, null);

if (getMatchingResourceResult.Success)
{
   if (getMatchingResourceResult.ResourceDescription != null)
   {
      // If there is a matching resource, just start to render it
      contentDistributionProxy.Play(
        getMatchingResourceResult.ResourceDescription.ResourceId,
        rendererId);
   }
}
else
{
   Console.WriteLine("Error: Gettting matching resources failed" +
     " with the following message: " + getMatchingResourceResult.Reason);
}
```

There are two variables, that have to be there in advance: contentId is the ID of a content object which could have been retrieved by a call to the *Find()* function like in the example of the respective section. The rendererId is the ID of a renderer device which could have been retrieved by listing all renderers using a call to *GetRenderers()* like in the first example. Having those two values, we can ask the service for a matching resource for the combination of the specified content ID and renderer ID. If the system can find a matching resource, we can then immediately start rendering it, by calling the *Play()* function with the returned resource ID. This

call will create a new session ID which can then be retrieved by getting a list of all sessions like in the example above.

## 7.10 Further examples and references

Additionally to the provided examples there is an implementation whose source code can be used as a living example: The ContentDistributionClient application which is a C# implementation and part of the source distribution of the Content Distribution installer.

For getting all the tools that are needed to compile and use the Content Distribution service see [MSOFTDN].

For a detailed discussion of the technology of Web Services see: [MSOFTWS].

The CC/PP specification can be found here [CCPP].

For getting access to the installation packages which also contain the source code for the service interface and the other three components see [AmigoRepository].

For a detailed overview of all the functions that are there in the content distribution interface, see the automatically generated SDK documentation that comes along with the installation package.

## 7.11 Resources

[AmigoRepository] Amigo Open Source Repository, http://amigo.gforge.inria.fr/home/index.html.

[CCPP] Composite Capabilities/Preferences Profile Home Page http://www.w3.org/Mobile/CCPP/

[Cyberlink] Cyberlink for Java, http://www.cybergarage.org/net/upnp/java/index.html

[MSOFTDN] Microsoft Download Center, http://download.microsoft.com

[MSOFTWS] Web Services and Other Distributed Technologies, http://msdn.microsoft.com/webservices

[MultimediaOWL] Service Description Vocabulary Ontologies, http://amigo.gforge.inria.fr/home/components/wp3/Ontologies_Vocabulary/download/index.html

[OWL] Web Ontology Language, 2004, http://www.w3.org/2004/OWL/

[UPnPAVCD] ContentDirectory:1 Service Template Version 1.01, June 25, 2002, http://www.upnp.org/standardizeddcps/documents/ContentDirectory1.0.pdf

# 8 How to use the Datastore to store persistent data within the Amigo network

## 8.1 Overview

### 8.1.1 Objectives and principles

The Datastore is a service that implements a simple persistency layer for storing data within the Amigo network. This allows all services/applications within the Amigo network that need to store data, just to use a common service, because storing of data is a need that most of the services will have.

The service is network based and can be used via its Web Service interface. To make use of it you need to create an application that can use Web Services and probably create a proxy object using the Datastore's WSDL document. The Datastore uses SQL Express as its backend and this should be installed and configured correctly with the alias "AmigoDatabase" pointing to a usable database.

*Hint:* Some implementations of Web Services calls restrain the size of Web Service call result. Due to the fact that this is a database implementation and the result of such a call can be very big, it can be necessary to increase this limit.

### 8.1.2 Features

The Datastore basically has the following functionalities:

- Create/manage compartments
- Modify data rows within a compartment
- Go back within history of a specific compartment/data row
- Send events about Datastore activities using WSEventing
- Automatic backups

Automatic backups are done by the Datastore service. In case of a database failure, these backups are then automatically restored. The database backup path and backup interval can be configured using the registry.

The following examples show how to work with the generated Datastore proxy. They assume that a proxy is already generated and connected to the service. The name of the proxy is assumed to be "dataStoreProxy". For examples how to discover the service see the .Net programming framework documentation.

### 8.1.3 Assessment

Quite some of the many services that exist in the Amigo context need to store persistent data. Having every single one of them to design and implement their own persistency layer would increase the overall system hardware and software requirements dramatically. The Datastore provides a solution for those services which need persistency for simple, non relational data. The Datastore service reduces the total number of dependents that all services within Amigo have, because the common need for a persistence layer is satisfied once.

## 8.2 How to create compartments

The following example shows how to add a new compartment to the Datastore. Every client application or service can have an arbitrary number of independent compartments to store their data in. A compartment is comparable to a data table in a database, but there are no Datastore level connections between those compartments like in a relational database.

```
DatastoreProxy .AddCompartmentResult addCompartmentResult =
  DatastoreProxy.AddCompartment(clientCredentials, "New Compartment", "DATACONTRACT", 10);

if (addCompartmentResult.Success)
{
    MessageBox.Show("Succesfully added compartment.");
}
else
{
    MessageBox.Show("Adding of compartment failed: " + addCompartmentResult.Reason);
}
```

As you can see, the result of the method AddCompartment() has some members. This is true for all functions of the Datastore interface. Additional to the expected result, there are always two more members: Success and Reason. Success is a Boolean value indicating if the call succeeded, or not. If Success has the value “false”, the call failed and the member Reason contains a text describing why, so it can be presented to the user as an error describing text. In this example, it is simply displayed in a message box.

To create a compartment we need four parameters: A client credentials object, a name for the compartment and a datacontract. The client credentials specify the name of the calling service to guarantee that there are no collissions with the compartment names of other services. The name of the new compartment must be unique, otherwise the call will fail. The next string specifies the data contract string, which must be replaced by a real datacontract xml document. These documents can be generated using the datacontract editor that comes along with the installation package of the service. An example data contract looks like this:

```
<?xml version="1.0"?>
<emicdc:DataContract xmlns:emicdc="DataContractSchema.xsd">
  <emicdc:Description>Music data contract</emicdc:Description>
  <emicdc:DataField>
    <emicdc:Name>Title</emicdc:Name>
    <emicdc:Type>TextField</emicdc:Type>
    <emicdc:Length>50</emicdc:Length>
  </emicdc:DataField>
  <emicdc:DataField>
    <emicdc:Name>Artist</emicdc:Name>
    <emicdc:Type>TextField</emicdc:Type>
    <emicdc:Length>50</emicdc:Length>
  </emicdc:DataField>
  <emicdc:DataField>
    <emicdc:Name>Year</emicdc:Name>
    <emicdc:Type>IntegerField</emicdc:Type>
  </emicdc:DataField>
  <emicdc:DataField>
    <emicdc:Name>Genre</emicdc:Name>
    <emicdc:Type>TextField</emicdc:Type>
    <emicdc:Length>30</emicdc:Length>
  </emicdc:DataField>
  <emicdc:DataField>
    <emicdc:Name>Price</emicdc:Name>
    <emicdc:Type>FloatField</emicdc:Type>
  </emicdc:DataField>
  <emicdc:DataField>
    <emicdc:Name>Article Number</emicdc:Name>
    <emicdc:Type>IntegerField</emicdc:Type>
  </emicdc:DataField>
  <emicdc:DataField>
    <emicdc:Name>Cover</emicdc:Name>
    <emicdc:Type>ByteField</emicdc:Type>
  </emicdc:DataField>
</emicdc:DataContract>
```

The datacontract is called “Music data contract” and you can see the specification of 7 fields. The field “Title” i.e. is specified as field with the type “TextField” with the length 50.

The last parameter of the AddDataContract() function is the versioning depth of the compartment. Each compartment can have its own history which can be used to undo changes made to its content. The versioning depth specifies how many steps for undoing will be saved for the compartment.

After the call the new compartment is ready to use, for an example see the next section.

## 8.3 How to work with compartments

Within a compartment, which essentially can be understood as a data table, the user can add, edit and delete data rows. Each operation on a specific data row will create a new version in the history of the given row, if a history depth > 0 was specified for the according compartment. The following example shows how some data rows are added to the compartment.

```
DatastoreProxy .AddDataRowsResult addDataRowsResult =
  DatastoreProxy.AddDataRows(clientCredentials, compartmentName, newRows);

if (addDataRowsResult.Success)
{
   // The result of the AddDataRows call is a list of added data rows
   // These can be used to determine which rows have been added.
}
else
{
   MessageBox.Show("Adding of datarows failed with the " +
            "following message: " + addDataRowsResult.Reason);
}
```

The code shows how a list of data rows is added to the compartment. This list of data rows is specified as a DataStoreResultSet object. The result of the AddDataRows operation is a list of data row IDs in the order of the rows in the DataStoreResultSet “newRows”.

The structure of a DataStoreResultSet and its dependent classes is the following:

```
public class DataStoreResultSet
{
   public string Name;

   public DataStoreResultRow[] Rows;

   public DataStoreResultColumn[] Columns;
}
public class DataStoreResultRow
{
   public DataStoreResultValue[] Values;
}
public partial class DataStoreResultColumn
{
   public string Name;

   public DataFieldType Type;
}
public enum DataFieldType
{
   TextField, IntegerField, FloatField, ByteField, DateTimeField
}
```

The DataStoreResultSet is a combination of a name for the result set, a list of columns and a list of rows. The columns define the structure and the content types for the rows. Each row must have a number of fields equal to the number of columns defined in the DataStoreResultSet. Each value in a row must have a type equal the according column

definition. Objects of the type DataStoreResultSet are returned when querying the Datastore for a list of data rows, adding data rows or listing the history.

One of the above added data rows should now be removed again. We assume we have the according data row id in the variable dataRowId.

```
DatastoreProxy .RemoveDataRowResult removeDataRowResult =
  DatastoreProxy.RemoveDataRow(clientCredentials, compartmentName, dataRowId);

if (removeDataRowResult.Success)
{
   Console.WriteLine("Datarow successfully deleted." );
}
else
{
   MessageBox.Show("Removing of datarow failed with the " +
         "following message: " + removeDataRowResult.Reason);
}
```

## 8.4 How to use the history

As explained in the example above each compartment can have its own history. The following example will show how to access and use the history of a compartment.

```
DatastoreProxy .ListHistoryResult listHistoryResult =
  DatastoreProxy.ListHistory(clientCredentials, compartmentName,
    dataRowId);

if (listHistoryResult.Success)
{
   // Do something with the history list, which is a
   // DataStoreResultSet object
}
else
{
   MessageBox.Show("Listing of history failed with the " +
         "following message: " + listHistoryResult.Reason);
}
```

This call to ListHistory() will list the whole content of the comparment containing all revisions of all datasets that exist within the history. The returned DataStoreResultSet object can then be inspected and all revisions up the the specified history depth can be seen. Every datarow has an implicit column called “id” whichs value can be used to access the datarow explicitly.

To revert the compartment to a point in history there are basically two functions: RevertDataRowByDate() and RevertDataRowBySteps(). The first one is used to revert the specified data into a state in which the datarow was at the specified point in time whereas the second function will revert the datarow by the number of steps in its history. The functions can be used like this:

```
DatastoreProxy.RevertDataRowByStepsResult revertDataRowByStepsResult =
  DatastoreProxy.RevertDataRowBySteps(clientCredentials, compartmentName, dataRowId, 3);

if (revertDataRowByStepsResult.Success)
{
   MessageBox.Show("Reverting the datarow was successful.");
}
else
{
   MessageBox.Show("Reverting of datarow failed with the " +
         "following message: " + revertDataRowByStepsResult.Reason);
}
```

This call will revert the dataset with the given dataRowId, which can be drawn from a call of “ListHistory()” i.e., 3 steps in history, that means the datarow is reverted to its state before the

last 3 changes. The parameters clientCredentials and compartmentName work like discussed in the first example.

## 8.5 Resources

For getting all the tools that are needed to compile and use the Datastore service see: http://download.microsoft.com

For a detailed discussion of the technology of Web Services see: http://msdn.microsoft.com/webservices

For getting access to the installation package which also contains the source code for the service see: http://amigo.gforge.inria.fr/

For a detailed overview of all the functions that are there in the Datastore interface, see the automatically generated SDK documentation that comes along with the installation package.

# 9 How to use home system deployment, configuration and management

## 9.1 Overview

Home system deployment, configuration and management stands at the crossing and is a common concern of different efforts within Amigo. We list below some of these efforts:

- Programming frameworks provide some service deployment and configuration capabilities.
- There is ongoing work on a management console for deploying software in the home and checking its status.
- Mature as well as ongoing work on ontologies and related tooling supports contextual models of the home.
- The security framework provides a base of identities and roles for users, devices, services.
- The User Modelling and Profiling Service (WP4) provides a base of profiles for users.
- The Context Management Service (WP4) provides actual contextual configurations of the home.

In this chapter, we present ongoing work that addresses the management console for software deployment in the home.

### 9.1.1 Objectives and principles

The management console is the single point of contact regarding service control and diagnostics for the home. For this, the management console relies on a (minimal) implementation on the client side with a standardized (Web Service) interface: the client deployment component. This component is installed during the bootstrapping phase of the device installation in an Amigo home and ideally pre-installed on an Amigo device. All additional software that is to be managed and diagnosed by the management console is installed on top of and by this deployment component.

In this way, the management console is able to communicate (in a standardized way) with the device to install/update and verify the state of software packages. Additionally, through this component, the management console is able to collect different kinds of information from the client (e.g. logging, performance and capacity information) and its installed software components.

Software component development is simplified by an SDK that enables the deployment component on the client to collect the required information from the software component. A schematic overview is given in Figure 9-1.

Since this document primarily targets the software developer, a preliminary version of the .Net SDK for client side software component development is described together with the components that are part of the .Net deployment client that will enable the management console to provide its intended functionality.

Additionally (also not directly intended for the target audience of this document), the Web Service interface used by the Management Console itself is described.

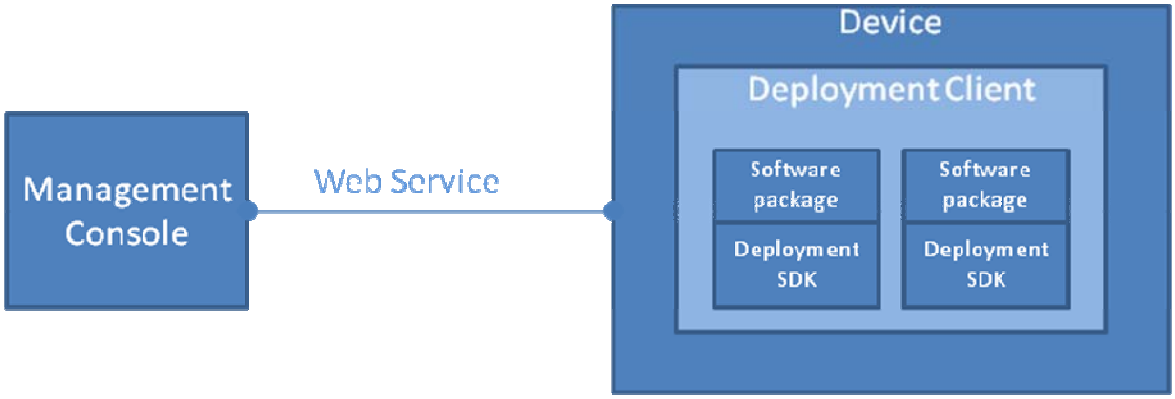


*Figure 9-1: A schematic overview of the management console and deployment client*

Note: the management console and client side deployment component are currently being developed and the described SDK and Web Service interface are likely to change in the near future.

### 9.1.2 Assessment

The Management Console provides a common interface for exposing home management tasks. These tasks include “first time” deployment of Amigo services in a platform independent way, software updates, and the diagnosis of both expected and unexpected problems.

The advantages of using the Management Console and supporting framework are that many of these “overhead” tasks are already taken care of. This frees the application developer to focus on the value-add of their application or service. The end user then has a simple and consistent access point for this functionality.

## 9.2 Platform support

The Management Console itself is implemented using the .Net programming framework and uses a Web Service interface towards the client deployment component. The client side might use an existing deployment component and SDK (e.g. OSGi) extended with a specific bundle that implements the Management Console Web Service interface (see How to use the management console interface) or use its own implementation (as is the case for .Net).

Separating the Management Console from the client deployment component by a Web Service interface ensures future compatibility and allows others to implement their own client components targeting or developed towards the specific needs of that platform.

## 9.3 Overview of the deployment client component (.Net)

The .Net deployment component is a component that needs to be installed on the client device. It hosts .Net software packages that are developed using the Deployment client SDK. For .Net, this component is installed as a Windows service.

### 9.3.1 Packages

.Net software packages intended to be hosted by the .Net deployment client component are called packages. These packages are containers for all the files (binaries, resources, configuration files, etc.) necessary for the software component at runtime. Packages have options describing their behaviour and can be digitally signed. The deployment client

component can be configured to either reject unsigned packages or deploy them in a sandboxed environment under restricted rights.

### 9.3.2 Package tool

Packages are created programmatically using the Packaging SDK or manually using the package tool. This tool allows a developer to create a package containing the files for his/her software component, set the necessary options and digitally sign the package.

## 9.4 How to use the deployment client SDK (.Net implementation)

### 9.4.1 Client side – deployment

.Net components that use the .Net deployment client must use the .Net deployment client SDK. This SDK simplifies many tasks for the software developer and allows the Management Console to control and collect information from it.

A simple example of a component using the SDK is given below:

```
using EMIC.DeploymentPlatform.Runtime;
…
{
   public class TestClass : Deployable
   {
      public override void OnInstallation()
      {
         this.IDeploymentContext.GetLogger().LogInformation("Installed");
      }

            public override void OnStart()
      {
         this.IDeploymentContext.GetLogger().LogInformation("Started");

      }

      public override void OnStop()
      {
         this.IDeploymentContext.GetLogger().LogInformation("Stopped");
      }


      public override void OnUninstall()
      {
         this.IDeploymentContext.GetLogger().LogInformation("Uninstalled");
      }
   }
}
```

The Deployable base class contains the core functionality for .Net deployment client software. The deployment component will call the respective functions during Installation, starting, stopping and uninstallation of the component.

Each function uses the IDeploymentContext interface to log its information. This information is then collected by the deployment client that can then in turn send it to the Management Console.

The code above serves as an example of a client component that uses the .Net deployment SDK. Further details and functionality will be described in the online documentation.

## 9.5 How to use the management console interface (Web Service)

The Management Console uses a standard Web Service interface to communicate with the deployment client. To enable the future integration of different platforms, the interface is kept language neutral.

The planned features require the interface to support deployment (installation/uninstallation/updating), control (start/stop), diagnose (per installed software component a status report, and a device status report).

Since the interface at this point in time is not finalized, the interface functions are shown in pseudo language.

### 9.5.1 Deployment

The deployment feature handles the installation, update and un-installation of software packages. To be platform independent, packages should be sent as a byte array. The result should indicate first of all success or failure. In case of failure, a predefined error code with a message should be returned.

```
DeploymentResult
{
        boolean success;
        ErrorCode errorCode;
        string errorDetails;
}

DeploymentResult = InstallPackage(byte[] package, string packageName);
```

Uninstallation takes a package name as a parameter and returns a similar result as the InstallPackage routine:

```
DeploymentResult = UninstallPackage(string packageName);
```

Update is similar to InstallPackage:

```
DeploymentResult = UpdatePackage(byte[] package, string packageName);
```

### 9.5.2 Control

The control feature enables the starting and stopping of software packages by the Management Console:

```
ControlResult
{
        boolean success;
        ErrorCode errorCode;
        string errorDetails;
}

ControlResult = StartPackage(string packageName);

ControlResult = StopPackage(string packageName);
```

### 9.5.3 Diagnose

There are two kinds of diagnostic functions. The first one addresses the installed software packages and returns information about their state. The second one addresses the deployment client/device and returns information about its state (loggings, memory, battery capacity, free disk space, etc.).

The information returned by both functions is in XML format. Since this functionality is under development only an example of such a logging can be given. The schema definition will be available when the component is first released.

Example logging:

```
<?xml version="1.0" encoding="utf-8"?>
<entries>
	<entry date="2/5/2007 8:11:21 PM">
		<type>information</type>
		<message>InstalledPackageRepository - creating.</message>
	</entry>
	<entry date="2/5/2007 8:11:21 PM">
		<type>information</type>
		<message>InstalledPackageRepository - created.</message>
	</entry>
</entries>
```

# 10 Conclusion

This deliverable has presented a set of HOWTOs with the aim of providing the Amigo developer with a means to quickly choose the most appropriate features of the Amigo middleware that assist with the development task at hand, and to familiarize the developer with how to use these features in the development of their own Amigo services and applications.

Chapter 2 covered how to use the OSGi and .Net programming frameworks provided by the Amigo middleware. Chapter 3 detailed how to secure access to the Amigo services in the home. Chapter 4 illustrated the range of options the Amigo developer has for describing semantic services, including simple, atomic capability descriptions, complex conversation-based capability descriptions, event-based service descriptions, as well as context-aware and quality-of-service descriptions. This chapter also covered the use of the Amigo semantic service repository. Chapter 5 demonstrated how the Amigo developer could exploit the suite of service discovery and service composition methods that the Amigo middleware offers in their applications, while domotic service development was covered in Chapter 6. For developers interested in developing applications that incorporate multimedia content, Chapter 7 provided a walkthrough of how to develop such an application. Using the persistent Datastore service was the focus of Chapter 8. Finally, Chapter 9 gave an overview of the Amigo home management console.

Having studied this deliverable, the Amigo developer will have the required knowledge and sufficient confidence to begin developing Amigo applications on their own. The Amigo middleware is an ongoing development however, and this deliverable will be updated in future revisions to include further enhancements made to the comprehensive suite of features offered by the Amigo middleware.